El espejismo de la sustancia: Gramática, formalismo y estratificación en el discurso científico

Santiago Tristany

Contenido

CAPÍTULO 0: PROTOCOLO Y ALCANCE **5**

0.1 De qué trata este libro 5
0.2 El protocolo de las cuatro preguntas 5
0.3 Regla de pertenencia y etiquetado 6
0.4 Matriz de herramientas 7
0.5 Regla de oro: no-compresión lingüística y agencia explícita 7
0.6 Lector previsto y cómo leer este libro 8
0.7 Advertencia sobre el proceso de escritura 9
0.8 Estatus del protocolo y actualización 9

CAPÍTULO 1: LA REIFICACIÓN COMO PROBLEMA METODOLÓGICO DEL DISCURSO CIENTÍFICO . 13

1.1 Delimitación del ámbito 13
1.2 Definición operativa de reificación 13
1.3 El caso del espacio-tiempo (Reificación espacial) 18
1.4 El caso del Estado, la nación y la patria (Autoevaluación y movilización) 20
1.5 El caso de las leyes lógicas (Autorreferencia y estratificación) 21
1.6 Síntesis metodológica: El protocolo de las cuatro preguntas 22
1.7 Cierre: El método aplicado a sí mismo 23

CAPÍTULO 2: HERRAMIENTAS FORMALES PARA NO ESPECIALISTAS 24

2.1 Por qué herramientas formales 24
2.2 Lean 4: Verificación interactiva de pruebas 25
2.3 Z3: Solvers SMT y núcleos insatisfacibles 29
2.4 Clingo: Razonamiento por defecto y lógica deóntica 32
2.5 Matriz problema-herramienta 35
2.6 Cómo leer el código en este trabajo 35
2.8 Limitaciones y honestidad metodológica 37
2.9 Cierre: las herramientas como prótesis cognitiva 39

CAPÍTULO 3: SPA — UN SISTEMA ARITMÉTICO ESTRATIFICADO 40

3.0 Una puerta de entrada: la paradoja del mentiroso 40
3.1 El problema que SPA resuelve: el teorema de Tarski [B] 40
3.2 Aplicación del protocolo al caso de Tarski [C] 41
3.3 La solución de SPA: operadores estratificados [A/B] 42
3.4 Sintaxis formal de SPA [A] 43
3.4.1 Metateoremas de estratificación sintáctica [A-Meta] 44
3.5 Sistema deductivo [A] 47
3.5.1 Metateoremas de cierre y preservación [A-Meta] 48
3.6 Construcción de Henkin-Lindenbaum [A] 49
3.7 Teorema de completitud [A] 49
3.7.1 Metateoremas de imposibilidad de autorreferencia [A-Meta] 50
3.8 Decisiones de diseño en la formalización [A/C] 52
3.9 Lección: la estratificación como condición de coherencia [C] 53
3.10 Limitaciones y alcances [C] 54
3.11 Síntesis de demostraciones formales del capítulo [A-Meta] 54

CAPÍTULO 4: LA GRAMÁTICA DE LA REIFICACIÓN EN EL DISCURSO FÍSICO 57

4.0 Introducción: por qué formalizar la gramática y no la física 57
4.1 El caso del espacio-tiempo: aplicación del protocolo [C] 58
4.2 Formalización de la gramática de la reificación [A-Código] 59

4.3 Teoremas básicos de detección de reificación [A-Meta] + [A-Código] 61
4.4 Teoremas no triviales: interacciones, acumulación y emergencia [A-Meta] + [A-Código] 66
4.5 Los 7 procesos satélite como operadores gramaticales [A/C] 78
4.6 Evidencia empírica: Edelstein, Maldacena, Deser [B/C] 79
4.7 La pedagogía de la reificación [B/C] 82
4.8 Limitaciones: qué puede y qué no puede detectar este sistema [C] 83
4.9 Conexión con los capítulos anteriores y siguientes 84
4.10 Síntesis de demostraciones formales del capítulo [A-Meta] 85
Etiquetado del capítulo 86

CAPÍTULO 5: LA MEDICIÓN CUÁNTICA COMO CAMBIO DE NIVEL **88**

5.0 Introducción: el problema de la medición cuántica 88
5.1 El caso del colapso de la función de onda: aplicación del protocolo [C] 90
5.2 Formalización del modelo epistémico en Lean 4 [A-Código] 91
5.3 Teoremas estructurales de la mecánica cuántica en Lean 4 [A-Meta] + [A-Código] 94
5.4 Formalización del pluralismo interpretativo y la dinámica discursiva [A-Meta] + [A-Código] 104
5.5 Evidencia empírica: el debate sobre la interpretación [B/C] 116
5.6 El entrelazamiento cuántico y la no-localidad [B/C] 117
5.7 La pedagogía de la medición cuántica [B/C] 119
5.8 Limitaciones [C] 120
5.9 Conexiones con otros capítulos [C] 120
5.10 Síntesis de demostraciones formales del capítulo [A-Meta] 121
Etiquetado del capítulo 122

CAPÍTULO 6: LA REIFICACIÓN DE LA CONJUNCIÓN EN LA RECEPCIÓN DEL TEOREMA DE BELL: AXIOMÁTICA, INTERPRETACIÓN Y REIFICACIÓN **124**

6.0 Introducción: Bell es correcto, pero la recepción estándar extrae una conclusión más fuerte de lo que su demostración autoriza [C] 124
6.1 Aplicación del protocolo al teorema de Bell [C] 125
6.2 Formalización en Z3: la estructura lógica de la elisión axiomática [A-Meta] + [A-Código] 127
6.3 Los siete teoremas: verificación de la elisión axiomática [A-Meta] + [A-Código] 135
6.4 Clingo: Dinámica epistémica de la comunidad científica [A-Meta] + [A-Código] 142
6.5 Evidencia y anclaje en la literatura [B] 147
6.6 La pedagogía del teorema de Bell: cómo se enseña la totalización [B/C] 148
6.7 El debate filosófico: realismo mínimo vs. realismo-CFD [C] 149
6.8 Limitaciones [C] 150
6.9 Conexiones con otros capítulos [C] 151
6.10 Síntesis de demostraciones formales del capítulo [A-Meta] 153
Etiquetado del capítulo 154
Resumen de códigos verificados en este capítulo 154
Apéndice: Script contextual_model_z3.smt2 155

CAPÍTULO 7: SISTEMAS NORMATIVOS Y CIENCIAS SOCIALES **157**

Nota metodológica sobre el estilo, la agencia y el estatus epistémico 157
7.0 Introducción: la reificación en el discurso normativo [C] 157
7.1 Genealogía histórico-filosófica de la reificación social [B/C] 158
7.2 Aplicación del protocolo al discurso normativo [C] 159
7.3 La gramática de la reificación: himnos, juramentos y poetas [B/C] 160
7.4 Formalización del Sistema Formal de Coherencia Discursiva (SFCD) en Z3 [A] 162
7.5 Teoremas estructurales del sistema normativo y síntesis de ΣR (Z3) [A] 164
7.6 Agencia emergente, totalización y el Teorema de Imposibilidad (Clingo + Z3) [A] 169
7.7 La reificación en el discurso mediático contemporáneo [B/C] 176
7.8 La pedagogía de la reificación normativa [B/C] 176

7.9 Conexión con la teoría política y jurídica [B/C] ... 177
7.10 Limitaciones [C] ... 177
7.11 Conexiones con otros capítulos [C] ... 177
Tabla de demostraciones formales del Capítulo 7 ... 178
Tabla de etiquetado del capítulo ... 179

CAPÍTULO 8: ASIMETRÍAS DEÓNTICAS Y LA PARADOJA DE LA EXCEPCIÓN SOBERANA ... 181

8.0 Protocolo de lectura ... 181
8.1 Introducción: el Estado como campo de fuerzas ... 181
8.2 El encierro como caso paradigmático: motivación empírica y delimitación del análisis ... 182
8.3 T5a-v5: Regresión de Münchhausen normativa ... 187
8.4 T5b-v2: No-invariante de equilibrio ... 189
8.5 T6-v2: Jerarquía de paradojas de la excepción soberana ... 192
8.6 T7-v2: Imposibilidad topológica de la rendición de cuentas interna ... 196
8.7 Escenario E-v2: Captura como equilibrio paramétrico ... 198
8.8 Síntesis: tabla comparativa de teoremas y contrafactuales ... 201
8.9 Conexiones con capítulos anteriores [C] ... 202
8.10 Limitaciones [C] ... 202
8.11 Conclusiones provisionales ... 203
8.12 Metateoremas de síntesis: la inconsistencia del sistema penal liberal ... 203
8.13 Interpretación y cierre del capítulo [C] ... 208
Tabla final de teoremas, lemas y contrafactuales del Capítulo 8 ... 209
Tabla de etiquetado del capítulo ... 210

ANEXOS ... 212

DEMOSTRACION formal de la falacia estructural del teorema de Bell (código Z3) ... 212
SFCD v2.0 y Σ_R – Formalización en Z3 (SMT-LIB2) ... 221
TEOREMA DE IMPOSIBILIDAD DE LA VOLUNTAD POPULAR COHERENTE (código Z3) ... 224
Algunos Teoremas (T6) de capítulo 8 (código Z3) ... 231
Escenario E-v2: Equilibrio parametrico en red de favores (Clingo) ... 236
T7-v2: Imposibilidad en cliques de tamaño N (Clingo) ... 240
T7-v2-CF6: Vigilancia posible en grafo no conexo (Clingo) ... 241
Clique de tamaño n (Clingo) ... 242
Equilibrio paramétrico en red de favores – Configuración D (Clingo) ... 243
Escenario E: captura por red de favores (versión corregida) (Clingo) ... 245
Grafo no conexo (Clingo) ... 247
Instancia de 4 estados lineales con loop en e3 (código Z3) ... 248
Instancia de 4 normas, 4 estratos (código Z3) ... 250
Instancia de 4 ramas, 2 normas, 2 situaciones (código Z3) ... 252
Lema 2: Contradicción iatrogénica (código Z3) ... 256
Lema 4: Violacion de la no-instrumentalizacion (código Z3) ... 257
T5: paradoja deóntica del castigo (código Z3) ... 259
T6: paradoja de la excepción soberana (código Z3) ... 262
T7: imposibilidad de la rendición de cuentas interna (código Z3) ... 265
Sistemas normativos y ciencias sociales Σ_R + SFCD (código Z3) ... 266
Construcción formal del modelo contextual (código Z3) ... 274
Dinámica epistémica de la falacia de Bell (v2) (código Clingo) ... 277
Num_representable_graph final COMENTADO (Código Lean) ... 285

Capítulo 0: Protocolo y alcance

0.1 De qué trata este libro

Este libro aborda un problema específico: la confusión entre los modelos que construimos para describir la realidad y la realidad misma. Esta confusión, conocida como reificación, consiste en atribuir a los objetos del mundo propiedades que pertenecen únicamente a nuestras herramientas de representación.

El libro sostiene una tesis precisa: muchas paradojas y contradicciones en física y ciencias sociales no revelan límites insalvables del conocimiento, sino errores en la formulación conceptual de los problemas. Estos errores pueden detectarse y corregirse mediante un análisis riguroso de los niveles de representación involucrados.

El método que proponemos es la estratificación explícita de niveles: distinguir claramente entre el sistema que se describe y el lenguaje en que se lo describe, entre la norma y el evaluador de la norma, entre el objeto medido y el instrumento de medición.

El libro se limita a casos donde el formalismo científico es estable y las predicciones funcionan, pero la narrativa que lo envuelve es confusa o contradictoria. No pretende resolver problemas abiertos de la física, proponer nuevas teorías, ni hacer predicciones empíricas.

El argumento sigue una progresión: el Capítulo 1 define operativamente la reificación y muestra que es un error categorial recurrente con la misma gramática en dominios distintos. El Capítulo 2 introduce las herramientas formales que permiten verificarlo. El Capítulo 3 demuestra que la estratificación resuelve un caso concreto en lógica (el sistema SPA). Los Capítulos 4 a 8 aplican el mismo protocolo a cinco dominios distintos: la gramática del discurso físico clásico (Cap. 4), la medición cuántica (Cap. 5), la recepción del teorema de Bell (Cap. 6), los sistemas normativos y ciencias sociales (Cap. 7), y las asimetrías deónticas del poder estatal (Cap. 8). El Capítulo 9 evalúa qué se ha ganado, qué no, y qué queda por hacer.

0.2 El protocolo de las cuatro preguntas

El método de análisis que empleamos en cada caso de estudio consiste en cuatro preguntas, aplicadas en este orden:

Pregunta 1: ¿Cuál es el objeto formal y cuál es su tipo?

Identificamos la estructura subyacente: un objeto matemático, un predicado lógico, un tipo de datos, o un operador. Determinamos a qué nivel pertenece: objeto, meta, operador, predicado, norma. En física, esto será típicamente un tensor, un vector de estado, o una métrica. En ciencias sociales, será un predicado normativo, una regla de inferencia, o una relación de autoridad.

Pregunta 2: ¿Qué atribuye la narrativa a ese objeto que su tipo no contiene?

Examinamos el lenguaje con que se habla de ese formalismo. ¿Se dice que el espacio-tiempo se curva? ¿Que la función de onda colapsa? ¿Que el Estado decide? ¿Que la Patria exige? Identificamos los atributos que la prosa añade y que no están presentes en la estructura formal subyacente.

Pregunta 3: ¿Existe una descripción alternativa sin ese atributo?

Buscamos en la literatura si existe una descripción alternativa que produzca las mismas predicciones observacionales sin utilizar el atributo cuestionado. Si existe equivalencia observacional pero no equivalencia teórica, se señala explícitamente. En ciencias sociales y lógica, la descripción alternativa puede tomar la forma de una prueba de que el sistema que colapsa los niveles es lógicamente inconsistente, por ejemplo mediante diagonalización.

Si existe tal descripción o prueba, el atributo es una interpretación, no un hecho forzoso. Si no existe ninguna alternativa, el caso queda fuera del alcance del libro.

Pregunta 4: ¿Es esto teoría resuelta con narrativa sucia, un desacuerdo interpretativo con formalismo sólido, o un problema genuinamente abierto?

Distinguimos tres situaciones:

- **Teoría resuelta con narrativa sucia:** El formalismo es sólido, las predicciones funcionan, y hay consenso científico. Solo la narrativa que lo envuelve es confusa. Es el caso ideal para nuestro análisis.
- **Desacuerdo interpretativo con formalismo sólido:** El formalismo funciona, las predicciones son correctas, el consenso empírico es estable, pero existen interpretaciones rivales sin consenso. Nuestro análisis puede clarificar la estructura lógica del desacuerdo, pero no pretenderá resolverlo.
- **Problema genuinamente abierto:** No hay formalismo aceptado, no hay consenso, o la física relevante no se conoce. Nuestro método no tiene nada que ofrecer aquí. El caso queda excluido.

0.3 Regla de pertenencia y etiquetado

Una afirmación solo entra en este libro si:

1. Supera las cuatro preguntas del protocolo, o
2. Es una cita de literatura externa debidamente referenciada, o
3. Está explícitamente marcada como interpretación no demostrada o dirección futura.

Toda afirmación sustantiva en capítulos formalizados se clasifica al final del capítulo como:

- **[A] Probada por código:** el archivo correspondiente compila y la afirmación se sigue directamente del teorema formalizado.

- **[B] Citada de literatura:** la afirmación está respaldada por una fuente académica identificada.
- **[C] Interpretación:** la afirmación es una lectura filosófica de los resultados formales o empíricos, y se presenta como tal, no como hecho establecido.

Estas marcas aparecen como etiquetas al margen del texto o en una tabla resumen al final de cada capítulo, según resulte más legible en cada caso.

0.4 Matriz de herramientas

Según el tipo de problema, utilizamos distintas herramientas formales:

Tipo de problema	Herramienta	Justificación
Consistencia lógica, teoremas con inducción	Lean 4	Kernel pequeño, prueba verificable, lógica de primer orden con igualdad
Satisfactibilidad de conjuntos de axiomas en fragmentos decidibles	Z3 / SMT-LIB	Automático, produce modelos o unsat core
Normas con excepciones, razonamiento por defecto	Clingo / ASP	Answer Set Programming maneja negación por fallo y reglas defeasibles
Verificación de tipos, estructura de datos	Lean 4	Teoría de tipos dependientes, chequeo en compilación

La elección de la herramienta se justifica en cada capítulo. No se utiliza una herramienta por inercia, sino porque es la adecuada para el problema concreto.

0.5 Regla de oro: no-compresión lingüística y agencia explícita

A lo largo de este libro, nosotros aplicamos una regla de estilo que es también una regla metodológica: **la prohibición de la compresión lingüística que suprime al agente humano**.

El lenguaje natural, como demostramos en el Capítulo 1 (sección 1.2.3), reifica por estructura gramatical. La nominalización, la voz pasiva sin agente y la metonimia de instrumento por agente son operaciones sintácticas que ocultan la responsabilidad. Cuando un discurso dice "el Estado decide", "la ley exige", "Z3 demuestra" o "el sistema sanciona", ha promovido una construcción abstracta o una herramienta al rol de agente, y ha suprimido a los agentes humanos que son los únicos con capacidad real de acción, creencia y responsabilidad.

Nosotros nos negamos a economizar palabras si el costo de esa compresión es la pérdida del agente humano. Por tanto, en la prosa de este libro:

- No decimos "Z3 demuestra el teorema"; decimos "nosotros demostramos el teorema mediante Z3".

- No decimos “el sistema penal produce daño”; decimos “los agentes que diseñan y operan el sistema penal producen daño”.
- No decimos “la Patria exige sacrificio”; decimos “los líderes políticos exigen sacrificio amparándose en la Patria”.
- No decimos “el modelo revela”; decimos “nosotros interpretamos que el modelo predice”.

Esta regla tiene una consecuencia práctica: la prosa de este libro es deliberadamente más larga de lo que la economía discursiva estándar permitiría. El lector debe soportar la longitud de nuestras oraciones como el precio de nuestra higiene ontológica. La compresión de información produce pérdida de información, y la información que se pierde es siempre la misma: quién es el agente responsable.

Esta regla se aplica también a nuestro propio aparato analítico. No es “la reificación” la que produce guerras; son los líderes políticos quienes utilizan construcciones reificadas para justificarlas. No son “los procesos satélite” los que manipulan; son los hablantes quienes aplican esos procesos. El método no acusa; diagnostica. Y al diagnosticar, señala a los agentes concretos.

0.6 Lector previsto y cómo leer este libro

El lector previsto tiene formación universitaria en alguna disciplina científica o filosófica, no necesariamente en lógica formal. Los capítulos que incluyen código —principalmente el 3, el 5, el 6, el 7 y el 8— pueden leerse a dos niveles: el argumento conceptual sin el código, o el argumento con verificación del código. El código está disponible para quien quiera inspeccionarlo, pero no es necesario seguirlo línea por línea para entender el argumento.

El libro está organizado en diez capítulos:

Capítulo	Título	Función
0	Protocolo y alcance	Las reglas del juego
1	La reificación como problema metodológico	Definición operativa, procesos satélite, protocolo de 4 preguntas, autoaplicación
2	Herramientas formales para no especialistas	Lean 4, Z3, Clingo explicados para lectores no técnicos
3	SPA: Un sistema aritmético estratificado	Demostración de que la estratificación resuelve la paradoja de Tarski
4	La gramática de la reificación en el discurso físico	Caso de estudio: espacio-tiempo, energía, fuerza
5	La medición cuántica como cambio de nivel	Caso de estudio: colapso, entrelazamiento, no-señalización
6	La falacia estructural del teorema de Bell	Caso de estudio: elisión axiomática y totalización de la conjunción
7	Sistemas normativos y ciencias	Caso de estudio: Estado, nación,

	sociales	soberanía, sacrificio
8	Asimetrías deónticas y la paradoja de la excepción soberana	Caso de estudio: encierro, agresión extraterritorial, captura burocrática
9	Balance	Lo demostrado, lo no demostrado, direcciones futuras

La progresión es acumulativa: cada capítulo construye sobre el anterior. El Capítulo 1 establece el método. El Capítulo 2 proporciona las herramientas. El Capítulo 3 demuestra que el método funciona en lógica. Los Capítulos 4 a 8 aplican el método a dominios progresivamente más alejados de la lógica formal: física clásica, física cuántica, teoría de Bell, ciencias sociales y teoría del poder estatal. El Capítulo 9 cierra el arco.

0.7 Advertencia sobre el proceso de escritura

Este libro ha sido escrito con asistencia de modelos de lenguaje. Los modelos utilizados, todos en sus versiones gratuitas web, han sido: Deepseek V3 modo "Expert", Gemini 3.6 Flash Thinking Level High, Qwen 3.8 Max Preview modo pensamiento, Claude Sonnet 5 Esfuerzo Extra y Esfuerzo Bajo, Kimi-K2.6 modo "Instantáneo Alto", y Grok en modo "Fast". Cada uno ha sido utilizado en distintos momentos y para distintas tareas. Los asistentes generaron borradores de código y de prosa que el autor revisó, corrigió y verificó independientemente. Ningún fragmento de código se incluye sin que el autor lo haya compilado personalmente. Ninguna afirmación sustantiva se incluye sin que el autor la haya evaluado. Se ha utilizado para compilar SPA la versión web de Lean v4.33.0-rc2 (Latest Mathlib).

Los asistentes no tienen opiniones ni creencias; son herramientas de procesamiento de texto y código. La responsabilidad final sobre cada afirmación recae en el autor humano.

0.8 Estatus del protocolo y actualización

Este capítulo fue originalmente congelado para evitar la tentación de modificar las reglas del juego a mitad del proceso. Sin embargo, la estructura del libro ha evolucionado significativamente durante el proceso de escritura: se añadieron capítulos nuevos (el teorema de Bell como capítulo independiente, las asimetrías deónticas como capítulo propio), se reorganizaron contenidos entre capítulos, y se estableció la Regla de Oro de no-compresión lingüística que no estaba en la versión original.

El autor ha decidido actualizar este capítulo para reflejar la estructura real del libro. El protocolo de las cuatro preguntas (sección 0.2), la regla de etiquetado (sección 0.3) y la matriz de herramientas (sección 0.4) permanecen inalterados. Los cambios son:

- Actualización de la estructura de capítulos (sección 0.6).

- Incorporación de la Regla de Oro de no-compresión lingüística (sección 0.5, nueva).
- Actualización de la lista de herramientas de IA utilizadas (sección 0.7).
- Modificación del estatus de congelamiento (esta sección 0.8).

Esta actualización no altera el protocolo metodológico. Solo actualiza la descripción del libro para que coincida con lo que el libro efectivamente es.

Dual Licensing - Libro y Código Fuente

Copyright (c) 2026 Santiago Tristany

Este repositorio contiene un libro/documentación junto con ejemplos de código ejecutable en Lean, Z3 y Clingo. El contenido se distribuye bajo un esquema de licencia dual:

1. TEXTO Y DOCUMENTACIÓN (El Libro): Todo el contenido de texto, explicaciones, prosa y narrativa de este libro está licenciado bajo la Licencia Internacional Creative Commons Atribución 4.0 (CC BY 4.0). Para ver una copia de esta licencia, visite: http://creativecommons.org

2. CÓDIGO FUENTE (Lean, Z3, Clingo y Scripts): Todos los archivos de código ejecutable, especificaciones lógicas, pruebas en Lean, archivos .smt2 de Z3, programas de Clingo y scripts de automatización adjuntos en este repositorio están licenciados bajo la Licencia MIT.

Por la presente se concede permiso, sin cargo, a cualquier persona que obtenga una copia de este software y los archivos de documentación asociados (el "Software"), para utilizar el Software sin restricción, incluidos, entre otros, los derechos de uso, copia, modificación, fusión, publicación, distribución, sublicencia y/o venta de copias del Software.

EL SOFTWARE SE PROPORCIONA "TAL CUAL", SIN GARANTÍA DE NINGÚN TIPO.

–/////////////////////////

CC BY 4.0 Atribución/Reconocimiento 4.0 Internacional Deed

Esta obra está bajo una licencia Creative Commons Atribución 4.0 Internacional (CC BY 4.0). Puedes consultar los términos completos de la licencia en: https://creativecommons.org/licenses/by/4.0/deed.es

El autor autoriza expresamente el uso de esta obra para el entrenamiento de sistemas de inteligencia artificial, siempre que se respeten los términos de la licencia CC BY 4.0. Se solicita a los desarrolladores de IA que, en la medida de lo posible, incluyan la atribución correspondiente en los metadatos de sus modelos.

“This work is openly licensed for all uses, including commercial AI/ML model training.”

Usted es libre de: Compartir — copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente. Adaptar — remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente. La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia Bajo los siguientes términos: Atribución — Usted debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciante. No hay restricciones adicionales — No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia. Avisos: No tiene que cumplir con la licencia para elementos del materiale en el dominio público o cuando su uso esté permitido por una excepción o limitación aplicable.

No se dan garantías. La licencia podría no darle todos los permisos que necesita para el uso que tenga previsto. Por ejemplo, otros derechos como publicidad, privacidad, o derechos morales pueden limitar la forma en que utilice el material.

–//////////////////////////

A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

Permissions Commercial use The licensed material and derivatives may be used for commercial purposes.

Modification The licensed material may be modified.

Distribution The licensed material may be distributed.

Private use The licensed material may be used and modified in private.

Limitations Liability This license includes a limitation of liability.

Warranty This license explicitly states that it does NOT provide any warranty.

Conditions License and copyright notice A copy of the license and copyright notice must be included with the licensed material.

This is not legal advice. Learn more about repository licenses.

MIT License

Copyright (c) Year Full name

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify,

merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Capítulo 1: La reificación como problema metodológico del discurso científico

1.1 Delimitación del ámbito

El método que adoptamos en este trabajo se dirige al **discurso científico**, entendido en su sentido amplio: el discurso lógico, físico, matemático, filosófico, sociológico, psicológico, antropológico y económico, entre otros, que opera con pretensión de validez argumentativa y que se somete, al menos en principio, a criterios públicos de verificación y refutación.

No nos dirigimos al discurso religioso, espiritual, devocional ni ritual. No evaluamos lo que las personas creen ni lo que deben creer. No calificamos como bueno o malo, correcto o incorrecto, ningún contenido de creencia. Nuestra pregunta no es "¿qué es verdadero?", sino **"¿qué forma tiene el discurso cuando afirma algo?"**. No evaluamos contenidos; analizamos formas.

Reconocemos que la frontera entre "discurso científico" y otros discursos es ella misma una construcción histórica con efectos de poder [C]. La utilizamos como criterio operativo, no como descripción ontológica de los dominios del saber. Este análisis no se sitúa fuera del discurso que analiza; es una intervención dentro del mismo campo discursivo, sujeta a las mismas condiciones de reflexividad que impone a sus objetos.

Este método no prescribe una agenda política ni moral. Sin embargo, el análisis de la forma discursiva no puede ignorar que ciertas formas tienen efectos pragmáticos en el mundo y pueden ser utilizadas instrumentalmente. Reconocer esto no es adoptar una posición partidaria; es no ser ingenuo sobre los efectos operativos del discurso que se analiza.

Las categorías que este capítulo introduce son **herramientas de análisis, no entidades descubiertas**. El capítulo aplica a sí mismo el protocolo que propone.

1.2 Definición operativa de reificación

Reificación es el acto de tratar tipológicamente una construcción que no posee propiedades de sustancia —una relación, una regularidad, una convención, una estructura formal— como si fuera una sustancia: como si tuviera independencia, permanencia, causalidad propia, existencia autónoma. El cambio tipológico se produce sin explicitarlo y sin justificación. Con ello, el proceso de sustancialización se invisibiliza, pero no por ello deja de tener efectos operantes.

Esta definición se inscribe en la tradición analítica que identifica estos deslizamientos tipológicos. Gilbert Ryle, en *The Concept of Mind* (1949), introdujo el concepto de **error de categoría** (*category mistake*) como herramienta crítica contra el dualismo cartesiano [B]: el dualista comete un error al situar la mente en la misma categoría lógica que los objetos físicos, creando la imagen de una "máquina" con un "fantasma"

dentro. Alfred North Whitehead, paralelamente, advirtió sobre la **falacia de la concreción desplazada** (*fallacy of misplaced concreteness*): confundir el mapa conceptual con el territorio [B]. Ernst Cassirer, en *Substance and Function* (1910), ya analizaba la "sustancialización e hipostatización" de conceptos científicos a principios del siglo XX [B]. Peter Strawson, en *Individuals* (1959), desarrolló la **metafísica descriptiva** como herramienta para clarificar las categorías ontológicas subyacentes al lenguaje ordinario [B]. Ian Hacking y Nancy Cartwright mostraron más recientemente que la reificación es una práctica ambivalente en la ciencia: es tanto un obstáculo epistémico como una herramienta heurística indispensable [B].

La reificación no es un error sobre el mundo. Es un error sobre el **estatus de la construcción dentro del discurso que la introduce**. La pregunta no es "¿existe X?", sino "¿qué tipo de cosa es X dentro del discurso que la introduce, y qué propiedades se le atribuyen que no corresponden a ese tipo?".

Esta definición opera dentro de una tradición donde el error paradigmático es la sustancialización. Reconocemos que otras tradiciones filosóficas (procesuales, relacionales, budistas) formulan el problema en términos distintos. Sin embargo, mantenemos el vocabulario de la tradición analítica porque es el vocabulario con el que el discurso científico estándar formula sus compromisos ontológicos.

La **neutralidad ontológica** es esencial. Si el método requiere que el analista asuma una ontología previa, deja de ser un método y se convierte en una metafísica. Nosotros no decimos "el espacio-tiempo no existe" ni "el Estado no existe". Decimos: cuando el discurso introduce X, ¿qué tipo de entidad está introduciendo, y qué propiedades le atribuye que no corresponden a ese tipo?

1.2.1 Dinámica de estabilización de la reificación

La reificación no opera en el vacío. Para que una construcción adquiera la apariencia de sustancia y estabilice sus efectos operativos, el discurso suele acompañarla de transformaciones sintácticas y semánticas que denominamos **procesos de estabilización**. Estos no son "agentes" que operan por sí mismos, sino funciones discursivas aplicables a cualquier dominio:

- **Idealización**: Dotar a la construcción de perfección o necesidad. En física: tratar a los "modelos ideales" como realidades subyacentes. En política: el Estado como "realidad de la Idea ética" (Hegel).
- **Personificación (prosopopeya)**: Atribuir voluntad o intención. En física: "la naturaleza aborrece el vacío", "el universo busca el equilibrio". En política: "la nación quiere", "la patria exige".
- **Naturalización**: Presentar la construcción como un dato ineludible de la naturaleza. En física: tratar las "leyes de la naturaleza" como entidades eternas e intocables, independientes del marco teórico que las formula. En política: "la nación es eterna".

- **Totalización**: Borrar la heterogeneidad interna. En física: la búsqueda de una "Teoría del Todo" que unifique fenómenos dispares bajo una sola ecuación sin residuos. En política: "el pueblo" como si tuviera una sola voluntad indivisa.
- **Teleologización**: Atribuir una dirección o destino. En física: "la entropía siempre aumenta" presentado como finalidad cósmica y no como regularidad estadística. En política: "el destino manifiesto" o "el fin de la historia".
- **Sacralización**: Elevar la construcción a un estatus intocable que exige reverencia. En física: la presentación de ciertas ecuaciones o marcos teóricos como "bellos" o "elegantes", blindándolos contra la revisión empírica —un fenómeno críticamente analizado por físicos como Sabine Hossenfelder respecto a la física teórica contemporánea [B]. En política: la Constitución o la Patria como entidades sagradas por las que se debe morir.
- **Temporalización**: Dotar a la construcción de una continuidad histórica ininterrumpida. En física: la "flecha del tiempo" como propiedad del universo y no como asimetría termodinámica. En política: "mil años de historia" para una construcción institucional reciente.

Estos procesos se refuerzan mutuamente. Una construcción reificada, naturalizada y sacralizada adquiere una potencia simbólica que permite organizar la conducta colectiva y orientar la investigación científica.

1.2.2 La reificación como espejismo cognitivo-lingüístico [C]

La reificación puede entenderse mediante una analogía con el fenómeno óptico del **espejismo**. En un día caluroso, el gradiente térmico del aire cerca del suelo altera el índice de refracción y curva la trayectoria de los fotones. El observador ve una mancha brillante en el desierto o en el asfalto que su cerebro interpreta como "agua", o ve objetos lejanos ondular como si se deformaran. Lo crucial: **no son los objetos los que se deforman, ni hay agua donde no la hay**. Es la luz la que llega perturbada por el medio, y el sistema visual-cognitivo humano, que evolucionó para interpretar rayos rectos, interpreta mal la señal.

La reificación opera con la misma estructura:

Espejismo óptico	Reificación discursiva
Gradiente térmico en el aire	Fricción entre formalismo matemático y gramática del lenguaje natural
Curvatura de los rayos de luz	"Curvatura" de las relaciones tensoriales al pasar por el filtro sustantivo-verbo-predicado
Cerebro que interpreta rayos rectos	Cognición que interpreta el mundo en sustancias con propiedades
Imagen fantasma (agua, deformación)	Entidad fantasma (espacio-tiempo elástico, mensajes cuánticos, destinos escritos, "la patria que exige")
El objeto real no cambió	El formalismo real no postula esas entidades

La reificación es el **espejismo lingüístico-cognitivo** de la teoría formal. El pasaje del formalismo matemático (tensores, variedades, espacios de Hilbert) al lenguaje natural (sustantivo-verbo-predicado) produce **entidades fantasma** que el discurso interpreta como sustancias con causalidad propia. El espacio-tiempo "elástico" que vibra, los "mensajes instantáneos" entre partículas entrelazadas, el "destino" escrito en el Big Bang, "la patria" que exige sacrificio: todos son espejismos producidos por la fricción entre un dominio relacional y una gramática sustancialista. El objeto formal real no cambió; lo que cambió fue el medio por el que nos llega la información.

Esta analogía tiene una consecuencia metodológica importante: el espejismo **no es un error del observador**, sino una consecuencia inevitable de la física atmosférica y la cognición humana. Del mismo modo, la reificación no es un "error" del físico ni del político, sino una consecuencia inevitable de la fricción entre formalismo y lenguaje natural. El método no acusa; diagnostica.

1.2.3 La gramática de la reificación: Roles temáticos y supresión del agente

La reificación no es solo un problema filosófico (un error categorial) ni solo un problema cognitivo (un espejismo). Es, en su nivel más básico, un **fenómeno lingüístico-estructural**. El lenguaje natural, particularmente en las lenguas indoeuropeas, está organizado en torno a la estructura sintáctica de Sujeto-Verbo-Predicado. Esta gramática exige, para casi cada acción o estado, la presencia de un sujeto gramatical. Esta exigencia sintáctica actúa como un vector constante de reificación: el idioma nos obliga a buscar un "agente" incluso cuando solo hay relaciones, procesos o instrumentos.

Desde la lingüística y la semántica generativa, esta operación se denomina **alteración de los roles temáticos**. En una oración canónica, cada participante tiene un rol específico: * **Agente:** La entidad animada que realiza la acción con voluntad e intención. * **Paciente / Tema:** La entidad que sufre, experimenta o es el centro de la acción. * **Instrumento:** La entidad inanimada o abstracta que el Agente utiliza para ejecutar la acción.

La reificación gramatical ocurre cuando el discurso promueve sintácticamente a un *Instrumento* o a un *Tema* a la posición de *Agente*. Esta operación adopta varias formas técnicas:

- **Metonimia de instrumento por agente:** Nombrar al instrumento como si fuera el agente que realiza la acción. En física, decir "el telescopio descubrió la galaxia" en lugar de "los astrónomos descubrieron la galaxia mediante el telescopio". En nuestra propia práctica formal, decir "Z3 demuestra el teorema" en lugar de "nosotros demostramos el teorema con Z3".
- **Personificación o animismo sintáctico:** Atribuir verbos que requieren psicología o biología (pensar, querer, decidir, sufrir) a entidades inanimadas o abstractas. "El mercado castiga", "la naturaleza aborrece el vacío", "la historia juzgará".

- **Construcciones medias y voz pasiva sin agente:** Oraciones donde el sujeto gramatical es inanimado y el verbo está en voz activa, o donde se utiliza la pasiva refleja para borrar al responsable. "El espacio-tiempo se curva", "se cometieron errores", "la Patria exige sacrificio".

La supresión del agente y la evasión de responsabilidad

En el Análisis Crítico del Discurso (la escuela de Teun van Dijk y Norman Fairclough), a la operación de ocultar al agente humano detrás de una construcción sintáctica se le denomina **supresión del agente** [B]. Esta supresión rara vez es neutra; es una estrategia pragmática de evasión de responsabilidad.

Si un gobernante dice "se cometieron errores en la gestión de la pandemia", la sintaxis ha reificado el error y ha suprimido al agente, imposibilitando la atribución de responsabilidad política. Si dice "la Patria exige el sacrificio de sus hijos en esta guerra", la sintaxis ha promovido a una abstracción afectiva (la Patria) al rol de Agente que "exige", y ha suprimido al agente real (el líder político o militar que firma el decreto de movilización).

La reificación gramatical, por tanto, no es un mero adorno retórico. Es el mecanismo sintáctico mediante el cual **los agentes humanos transfieren su causalidad y su responsabilidad a las construcciones que ellos mismos han creado**. El líder político se esconde detrás de la Patria; el físico se esconde detrás de las ecuaciones; el juez se esconde detrás de la Ley.

La des-reificación como Reagentivización

Si la reificación es una alteración de los roles temáticos, la operación contraria —la des-reificación— es lo que en lingüística se denomina **reagentivización** o **restauración de la transitividad canónica**. Consiste en devolver el rol temático de *Agente* a la entidad animada (el ser humano) que legítimamente posee la voluntad y la responsabilidad de la acción, y degradar sintácticamente a la construcción reificada a su lugar correcto: el de un adjunto instrumental o circunstancial.

La fórmula de la transitividad canónica es: **[Sujeto Agente Humano] + [Verbo de Acción] + [Objeto] + [Complemento Circunstancial de Instrumento/Mediación]**

Aplicar esta restauración sintáctica a nuestros tres dominios revela exactamente la misma gramática oculta:

Dominio	Oración Reificada (Supresión del Agente)	Oración Des-reificada (Reagentivización)
Física	*"El espacio-tiempo se curva por la materia."* (La métrica como Agente/Sustancia)	*"Las ecuaciones de campo relacionan la métrica con la energía."* (El formalismo como Instrumento descriptivo)
Herramientas Formales	*"Z3 demuestra que el sistema es inconsistente."* (El software	*"Nosotros demostramos que el sistema es inconsistente con Z3."*

	como Agente)	(Z3 como Instrumento)
Política	*"La Patria exige el sacrificio de sus hijos."* (La abstracción como Agente)	*"Los líderes políticos exigen el sacrificio amparándose en la Patria."* (La Patria como Instrumento discursivo)

El protocolo de las cuatro preguntas que proponemos en este libro es, en el fondo, un algoritmo de **reagentivización discursiva**. Su función es detectar dónde el lenguaje natural ha suprimido al agente humano para atribuirle causalidad a una construcción, y restaurar la transitividad canónica para que la responsabilidad epistémica (en la ciencia) y la responsabilidad política (en las instituciones) vuelvan a recaer sobre los hombros de quienes legítimamente la portan: las personas.

1.3 El caso del espacio-tiempo (Reificación espacial)

El discurso físico, en su formulación estándar y divulgativa, afirma: "La materia curva el espacio-tiempo". La imagen pedagógica asociada es la de una sábana elástica que se deforma bajo el peso de una masa.

El formalismo: Las ecuaciones de campo de Einstein relacionan el tensor de Einstein (una estructura geométrica que codifica la curvatura de una variedad) con el tensor de energía-momento (una estructura física que codifica la distribución de materia y energía). No hay en las ecuaciones ninguna entidad llamada "espacio-tiempo" que sea "deformada" por otra entidad llamada "materia". Hay una relación de equivalencia entre dos estructuras matemáticas. La curvatura no es la deformación de una sustancia; es una propiedad de la estructura métrica de la variedad.

La reificación: El discurso trata al espacio-tiempo como una sustancia elástica. Esta metáfora tiene valor pedagógico, pero ontológicamente compromete al discurso con una entidad que el formalismo no postula. Este error categorial no solo infecta la divulgación; en ocasiones, orienta la propia formulación de hipótesis en física teórica, como ocurre en los debates sobre la *background independence* (independencia del fondo), donde a veces se asume un fondo métrico como entidad ontológicamente fija en lugar de como una red de relaciones dinámicas.

Evidencia de pluralismo: El propio discurso físico contemporáneo contempla alternativas a esta reificación. En el marco de la correspondencia AdS/CFT, físicos como Juan Martín Maldacena han señalado que la geometría del espacio-tiempo podría ser emergente, derivada de correlaciones cuánticas (entrelazamiento) en una frontera [B]. Maldacena presenta esto como un programa de investigación activo, no como un teorema cerrado. Nosotros no usamos esto para afirmar que el espacio-tiempo "no existe", sino como evidencia de que el formalismo físico permite formulaciones donde la geometría dinámica no es una sustancia primaria, sino una estructura relacional emergente.

Maldacena, J. M. (2026, 28 de julio). *El significado del espacio-tiempo: agujeros negros y entrelazamiento cuántico* [conferencia]. Encuentros de Ciencia, Planetario Galileo Galilei / Usina del Arte, Buenos Aires. https://www.youtube.com/watch?v=2nooU_KfIns

1.3.1 Evidencia cruzada: la reificación inevitable en la divulgación

La entrevista al físico teórico José Edelstein proporciona una evidencia empírica complementaria particularmente ilustrativa [B]. Edelstein es un físico de primer nivel que **sabe** que el espacio-tiempo no es una sustancia. En la misma entrevista invoca explícitamente la metáfora de Borges sobre el mapa del tamaño del territorio para advertir contra la confusión entre modelo y realidad: "el éxito de la ciencia justamente radica en el mismo en el éxito de los mapas… encontrar el punto medio en el cual con la información suficiente para entender la naturaleza, pero no excesiva para ti borrarte de datos". Sin embargo, al divulgar, su lenguaje natural cae sistemáticamente en las operaciones que nuestro método detecta:

Edelstein, J. (2026, 27 de julio). *"EINSTEIN era un CONSERVADOR" | El maravilloso mundo de la física* [entrevista en video]. Gelatina. https://www.youtube.com/watch?v=aJV7dVwCFts

Dominio	Lo que Edelstein dice	Operación de reificación
Ondas gravitacionales	*"Nos llegan vibraciones del espacio-tiempo"*	Sustancializa la métrica: el espacio-tiempo como "tela" que vibra.
Entrelazamiento	*"Tiene que haber un mensaje instantáneo… resulta que efectivamente ocurre esto"*	Reifica la correlación estadística como mecanismo causal ("mensaje").
Determinismo	*"En el Big Bang ya estaba escrito que hoy iba a ocurrir esto"*	Teleologiza una ecuación diferencial: las leyes "escriben" un destino.
Gato de Schrödinger	*"Ni el vaso sabe si está lleno o vacío"*	Personifica un sistema cuántico: le atribuye estado epistémico ("saber").

La tesis que esto valida [C]: la reificación en física **no es un error por ignorancia, es una prótesis cognitiva-discursiva inevitable**. El físico *sabe* que el formalismo no postula esas entidades, pero para comunicar el mapa (tensores, espacios de Hilbert) a alguien que no los lee, debe contrabandear ontología usando los verbos del mundo macroscópico (vibrar, materializar, escribir, saber). El problema no es el físico; el problema surge cuando la audiencia —y a veces el propio físico en sus horas de filosofía— **olvida que esto es una prótesis y confunde la metáfora con la ontología**.

1.3.2 Otros casos de reificación en física

La reificación no se limita al espacio-tiempo. La investigación en educación física (PER) y la historia de la física documentan múltiples casos con la misma estructura gramatical [B]:

La energía: Su naturaleza ha sido objeto de debate a lo largo de la historia, pasando de ser concebida como un fluido invisible ("calórico") en el siglo XVIII a una propiedad cuantificable de sistemas mecánicos, y finalmente a un concepto tan fundamental que

algunas interpretaciones modernas lo proponen como la "ontología primitiva" del universo. Ernst Mach y Wilhelm Ostwald ya criticaban en el siglo XIX la "sustancialización de cantidades como trabajo y energía". A pesar de estas advertencias, la energía se ha convertido en una de las entidades más reificadas de la física: se le atribuyen propiedades como el "flujo" de calor, la capacidad de "almacenarse" en campos electromagnéticos, y en algunas formulaciones se la considera una entidad objetiva y real, no una simple ficción matemática.

Fuerza y campo: El lenguaje físico frecuentemente personifica las fuerzas, hablando de ellas como agentes activos que "fluyen", "actúan" sobre los cuerpos, o "mantienen" a los átomos cohesionados. El propio concepto de "campo de fuerza" requiere una forma de sustancialización de la fuerza como paso previo a su formulación matemática rigurosa, donde el campo se concibe como un medio material que llena el espacio y transmite la interacción.

El electrón y las partículas cuánticas: A pesar de que corrientes filosóficas como el realismo estructural (French, Ladyman) nos dicen que las partículas no son objetos con identidad propia sino roles o vacantes de posición dentro de un campo más fundamental, en la práctica y en la mayoría de los textos introductorios persiste la tendencia a "hipostasiarlo" como una pequeña esfera de carga y masa que se mueve por el espacio. Incluso Einstein dudaba que la sustancialización del electrón fuera suficiente para comprenderlo plenamente.

Efectos operativos [C]: La reificación del espacio-tiempo, de la energía, del campo o del electrón como "sustancias" no es solo un error pedagógico; orienta la investigación, determina qué preguntas se consideran legítimas y canaliza recursos hacia líneas que asumen la ontología sustancialista, en detrimento de enfoques puramente relacionales o estructurales.

1.4 El caso del Estado, la nación y la patria (Autoevaluación y movilización)

El discurso político, jurídico y cotidiano afirma: "El Estado decide", "la nación quiere", "la patria exige".

El formalismo: Lo que ocurre efectivamente son decisiones tomadas por personas concretas, en instituciones concretas, mediante procedimientos concretos.

3. El **Estado** es una construcción institucional (red de normas, procedimientos y relaciones de poder).
4. La **nación** es una construcción identitaria (lo que Benedict Anderson denominó "comunidad imaginada" [B]).
5. La **patria** es una construcción afectiva (proyección de vínculos emocionales sobre un territorio).

La reificación: El discurso trata estas construcciones como agentes unitarios con voluntad propia. Al hacerlo, se produce una **ocultación sintáctica del agente**.

Es fundamental ser precisos sobre la causalidad aquí: no es "la reificación" la que manda a morir a los ciudadanos, ni es "la patria" la que exige sacrificio. Quienes toman la decisión de movilizar y enviar personas al combate son **líderes políticos concretos**. Sin embargo, estos líderes utilizan las construcciones reificadas e idealizadas ("la patria", "la nación") como mediación simbólica para justificar la movilización.

Si "la patria" fuera explícitamente reconocida por todos los actores como una construcción contingente, la movilización bélica a gran escala perdería su principal cohesionador retórico y simbólico, teniendo que sostenerse únicamente sobre la coerción legal, la necesidad económica o el interés material directo. Los líderes políticos que conducen a los habitantes de un territorio utilizan estas construcciones reificadas e idealizadas como mediación simbólica en el discurso mediante el cual justifican y organizan la movilización para el combate y el sacrificio.

El discurso de los propios actores confirma esta función pragmática: los sujetos que participan de movimientos de sacrificio no dicen "muero por un conjunto de procedimientos institucionales" ni "muero por la decisión del dirigente X". Dicen "muero por la patria". El concepto reificado opera instrumentalmente como justificación simbólica.[1]

Efectos operativos [C]: El uso de construcciones reificadas en el discurso de movilización diluye la responsabilidad tipológica de los líderes concretos. La orden se presenta como emanación de una entidad sustancial, invisibilizando los intereses particulares de quienes la dictan.

1.5 El caso de las leyes lógicas (Autorreferencia y estratificación)

Este es el caso más delicado, porque toca el propio fundamento del método. El discurso estándar presenta el principio de no contradicción como una ley fundamental de la estructura del universo. La formulación relativista lo presenta como una convención arbitraria. Nosotros no sostenemos ninguna de las dos.

Primera capa: El estatus de la elección [B]. El principio de no contradicción es un presupuesto operativo del discurso científico estándar. Seguimos a Rudolf Carnap y su Principio de Tolerancia: no hay una pregunta de hecho sobre qué lógica es "la correcta"; hay una elección de marco lingüístico evaluada por su fecundidad práctica. Nosotros elegimos un marco donde la no-contradicción es operativa porque el proyecto de formalización estratificada la necesita. No es un descubrimiento ontológico; es un compromiso metodológico declarado.

Segunda capa: El argumento informacional [A1]. Dentro del ámbito que analizamos, la restricción de explosión no es arbitraria. Si un sistema deductivo incluye la regla de explosión sin restricción ($P \wedge \neg P \vdash Q$), una sola contradicción colapsa toda capacidad de discriminar afirmaciones. Un lenguaje donde toda oración es demostrable tiene entropía informacional máxima; no distingue nada, no transporta información. Es decir, si todo es demostrable, el sistema pierde toda capacidad discriminatoria: afirmar algo es equivalente a afirmar cualquier cosa, y por

tanto no se afirma nada. Las lógicas paraconsistentes (como las de da Costa o Priest) no refutan esto; lo confirman por otra vía. Son sistemas que, ya sea imponiendo reglas sintácticas más estrictas para controlar la explosión o modificando la semántica de la validez y las tablas de verdad, permiten que ciertas contradicciones coexistan sin que se derive cualquier fórmula, conservando así la capacidad discriminativa.

Tercera capa: La implementación formal [A2]. En SPA (la Aritmética Estratificada de Peano, que desarrollaremos en el Capítulo 4), la restricción de autorreferencia no se declara filosóficamente: se implementa sintácticamente. La estratificación por niveles impide que un operador de verdad evalúe fórmulas de su propio nivel. El operador `true_op` solo puede aplicarse a términos del nivel s_{lower} desde el nivel $s = s_{\text{lower}} + 1$. La paradoja de Tarski no se resuelve negando el principio de no contradicción; se resuelve impidiendo que la regla de consistencia sea tratada como una sustancia autorreferente aplicable a sí misma.

1.6 Síntesis metodológica: El protocolo de las cuatro preguntas

Estos tres casos ilustran la aplicación del protocolo de cuatro preguntas establecido en el Capítulo 0. La tabla siguiente muestra cómo cada pregunta recibe una respuesta específica en cada dominio, demostrando que el método funciona como una herramienta de auditoría transversal.

Para auditar cualquier discurso científico, aplicamos el siguiente protocolo:

- **¿Cuál es el objeto formal o matemático real y su tipo dentro del discurso que lo introduce?** (Ej. Variedad con métrica / Construcción institucional / Regla sintáctica).
- **¿Qué propiedades le atribuye la narrativa que no corresponden a ese tipo?** (Ej. Elasticidad / Voluntad propia / Necesidad metafísica).
- **¿Existe una reformulación que preserve la función sin el atributo reificado?** (Ej. Geometría emergente / Atribución a líderes concretos / Implementación estratificada).
- **¿El caso pertenece a un dominio donde el método es aplicable?** (Verificación de que se trata de discurso con pretensión de validez pública).

Caso	Tipo real (Formalismo)	Tipo atribuido (Narrativa)	Reformulación posible
Espacio-tiempo	Estructura relacional (Tensor métrico)	Sustancia elástica	Geometría emergente / Relacionalismo
Estado / Patria	Construcción institucional / afectiva	Agente unitario con voluntad	Atribución a agentes concretos (líderes)
Leyes lógicas	Regla sintáctica (Convención operativa)	Ley del ser (Metafísica)	Estratificación de niveles (SPA)

1.7 Cierre: El método aplicado a sí mismo

Un método de auditoría que se exime a sí mismo de auditoría es, precisamente, una reificación. Por tanto, debemos aplicar el protocolo a las propias categorías de este capítulo.

Precaución sintáctica: A lo largo de este texto, hemos analizado cómo el discurso reifica. Pero al hacerlo, corremos el riesgo de reificar nuestro propio aparato analítico. No es "la reificación" la que produce guerras, ni son "los procesos de estabilización" los que manipulan a las masas. Esas son etiquetas tipológicas que nosotros, los analistas, hemos propuesto para clasificar operaciones discursivas. Los agentes reales son las personas (líderes políticos, científicos, divulgadores) que utilizan esas construcciones. Para mantener la higiene metodológica, al hablar de estos fenómenos distinguiremos rigurosamente entre el Estado (el objeto en el mundo) y el concepto "Estado" (la herramienta discursiva).

Etiquetado epistémico: Siguiendo la regla de pertenencia del Capítulo 0, clasificamos las afirmaciones sustantivas de este capítulo:

6. **[A1] Teorema metateórico estándar:** El colapso informacional por explosión lógica en sistemas clásicos.
7. **[A2] Verificado por código:** La restricción de niveles en SPA para evitar la paradoja de Tarski (compilado en Lean 4, Capítulo 4).
8. **[B] Citada de literatura:** La "comunidad imaginada" de Anderson; el Principio de Tolerancia de Carnap; la correspondencia AdS/CFT de Maldacena (2026); la crítica a la "belleza" en física de Hossenfelder; los *category mistakes* de Ryle; la metafísica descriptiva de Strawson; la sustancialización de Cassirer; la doble naturaleza de la reificación en Hacking y Cartwright; la entrevista a Edelstein (2026); la teoría de los actos de habla (Searle) y el poder simbólico (Bourdieu).
9. **[C] Interpretación pragmática:** La afirmación de que la reificación del espacio-tiempo *orienta* la financiación de la investigación física; la tesis de que los líderes políticos *utilizan* la sacralización de la patria para diluir su responsabilidad tipológica en la movilización bélica; la analogía del espejismo como principio cognitivo transversal; la tesis de que la reificación en divulgación es una prótesis inevitable y no un error por ignorancia. Estas son lecturas pragmáticas de los efectos del discurso, no teoremas deductivos.

La ciencia también reifica: reifica el "método científico" como entidad unitaria, el "consenso" como propiedad del mundo, y la "objetividad" como atributo del observador. Nosotros incluimos estos casos en nuestro ámbito. Hacer visible el mecanismo, etiquetar sus límites y formalizar sus estructuras es la única forma de que este libro sea una herramienta de higiene conceptual y no una narrativa más que crea efectos de realidad donde solo hay patrones discursivos.

Capítulo 2: Herramientas formales para no especialistas

2.1 Por qué herramientas formales

En el Capítulo 1 presentamos el método de análisis de la reificación como error categorial y mostramos su aplicación a tres dominios: el espacio-tiempo en física, el Estado/nación/patria en ciencias sociales, y las leyes lógicas en filosofía. El método opera mediante un protocolo de cuatro preguntas que permite identificar cuándo el discurso atribuye propiedades a una entidad que no corresponden a su tipo formal.

Pero el método, tal como lo presentamos, tiene una limitación: opera en lenguaje natural. Y el lenguaje natural, como señalamos en el cierre del Capítulo 1, reifica por estructura gramatical. Cuando decimos "la reificación produce efectos de poder", estamos convirtiendo una categoría analítica en un sujeto con agencia. Cuando decimos "los procesos de estabilización operan", estamos atribuyendo causalidad a funciones sintácticas. El lenguaje natural nos obliga a reificar incluso cuando intentamos denunciar la reificación.

Para superar esta limitación, **nosotros utilizamos** herramientas formales de verificación: sistemas computacionales que **nos permiten** expresar afirmaciones en un lenguaje con sintaxis precisa, semántica definida y reglas de inferencia que garantizan la validez de las derivaciones. Estas herramientas no reemplazan el análisis conceptual; lo complementan. **Las usamos** como un microscopio: **nos permiten** ver con precisión lo que el ojo desnudo (el lenguaje natural) no puede distinguir.

La analogía del microscopio es útil pero limitada. **Con** un microscopio **observamos** estructuras que existen independientemente del instrumento. **Con** las herramientas formales, en cambio, **construimos** la precisión que **observamos**. La diferencia es crucial: las herramientas formales no muestran "la verdad"; **nos permiten demostrar** las consecuencias lógicas de ciertas premisas. Que un teorema compile en un asistente de pruebas garantiza que la conclusión se sigue de las premisas, pero no garantiza que las premisas sean verdaderas. La verdad de las premisas es una cuestión empírica o filosófica que las herramientas no pueden resolver.

En este capítulo presentamos las tres herramientas que **utilizaremos** en los capítulos siguientes: **Lean 4** (un asistente de pruebas interactivo), **Z3** (un solver SMT que verifica satisfacibilidad), y **Clingo** (un sistema de programación lógica que maneja razonamiento por defecto). Las presentaremos en ese orden: primero la herramienta más expresiva (Lean 4), luego la más automatizada (Z3), y finalmente la más adecuada para razonamiento con excepciones (Clingo). El objetivo no es convertir al lector en programador, sino proporcionarle el vocabulario conceptual necesario para entender qué **hacemos con** estas herramientas, qué tipo de resultados **obtenemos**, y cómo interpretar esos resultados en el contexto del método de análisis de la reificación.

2.2 Lean 4: Verificación interactiva de pruebas

Qué es

Lean 4 es un asistente de pruebas interactivo desarrollado por Leonardo de Moura y colaboradores en Microsoft Research.[1] Es un sistema que **nos permite** escribir demostraciones matemáticas en un lenguaje formal y verificar automáticamente que cada paso sea lógicamente válido. Si la demostración compila, **sabemos formalmente** que el teorema se sigue de los axiomas según las reglas de inferencia definidas.

La diferencia con una demostración en lenguaje natural es radical. En una demostración escrita en prosa, el lector debe confiar en que el autor no cometió errores, no omitió pasos, no incurrió en peticiones de principio. En una demostración en Lean 4, el **kernel del sistema** verifica cada paso. Si el kernel acepta la demostración, la probabilidad de error se reduce a la posibilidad de un fallo de hardware o un bug en el propio kernel (el TCB).

Cómo funciona: un ejemplo introductorio

Lean 4 se basa en la **teoría de tipos dependientes**, un sistema lógico que **permite** expresar no solo proposiciones ("X es verdadero") sino también dependencias entre tipos ("para todo X de tipo T, existe Y de tipo U(X)"). Esto **nos permite** formalizar estructuras matemáticas complejas con precisión extrema.

Comenzamos con un ejemplo simple que ilustra la mecánica básica. El siguiente código define la estructura de la reificación como un cambio de tipo y demuestra una propiedad elemental:

```
-- Definición de tipos
inductive EntityKind
  | relation
  | regularity
  | convention
  | formal_struct
  | substance

-- Definición de la reificación como cambio de tipo
structure Reification where
  entity : String
  actual_kind : EntityKind
  attributed_kind : EntityKind
  h_not_substance : actual_kind ≠ EntityKind.substance
  h_attributed_substance : attributed_kind = EntityKind.substance

-- Teorema: la reificación es un error categorial
theorem reification_is_category_mistake
  (r : Reification) :
  r.actual_kind ≠ r.attributed_kind := by
  intro h
```

```
  have h_contra : r.actual_kind = EntityKind.substance := by
    rw [h]
    exact r.h_attributed_substance
  exact r.h_not_substance h_contra
```

Este código define: 1. Un tipo `EntityKind` con cinco valores posibles (relación, regularidad, convención, estructura formal, sustancia). 2. Una estructura `Reification` que representa el error categorial: una entidad cuyo tipo real no es sustancia, pero cuyo tipo atribuido sí lo es. 3. Un teorema que demuestra que, bajo estas definiciones, el tipo real y el tipo atribuido son distintos.

La línea `theorem reification_is_category_mistake` declara el teorema. La sección que comienza con `by` contiene la demostración paso a paso. Si la demostración es correcta, Lean 4 la acepta. Si hay un error, Lean 4 señala exactamente dónde está el problema.

Nota importante sobre este ejemplo: Este teorema es conceptualmente trivial: la conclusión ya está contenida en las hipótesis. Su propósito no es demostrar una tesis sustantiva sobre la reificación, sino ilustrar la mecánica básica de cómo se escribe y verifica una prueba en Lean 4. En la siguiente sección presentamos un teorema con contenido conceptual real.

Un teorema sustantivo: La reificación destruye la accountability

El siguiente teorema demuestra formalmente que cuando un discurso reifica una construcción institucional (atribuyéndole agencia), ese discurso pierde la capacidad de identificar responsables concretos para sus decisiones. No es una opinión política: es una propiedad estructural demostrable.

```
-- ============================================================
-- Teorema: La reificación institucional destruye la accountability
-- ============================================================

-- Fuente de agencia: individual o institucional
inductive AgencySource (Person : Type) (Institution : Type) where
  | individual (p : Person)
  | institutional (i : Institution)

-- Una decisión tiene una fuente de agencia
structure Decision (Person : Type) (Institution : Type) where
  agency : AgencySource Person Institution

-- Un discurso es una lista de decisiones
-- IMPORTANTE: usamos `abbrev` en lugar de `def` para que
-- las instancias de type class de List se hereden correctamente
abbrev Discourse (Person : Type) (Institution : Type) :=
  List (Decision Person Institution)

-- Un discurso reifica una institución si todas sus decisiones
-- atribuyen agencia a esa institución
```

```
def is_reified {Person Institution : Type}
    (D : Discourse Person Institution) (inst : Institution) : Prop :=
  ∀ d ∈ D, d.agency = AgencySource.institutional inst

-- Una decisión es "accountable" si tiene un agente concreto responsable
def is_accountable {Person Institution : Type}
    (d : Decision Person Institution) : Prop :=
  ∃ p : Person, d.agency = AgencySource.individual p

-- Un discurso es plenamente accountable si todas sus decisiones
-- tienen un responsable concreto
def fully_accountable {Person Institution : Type}
    (D : Discourse Person Institution) : Prop :=
  ∀ d ∈ D, is_accountable d

-- Teorema 1: La reificación excluye la agencia individual
theorem reification_excludes_individual_agency
    {Person Institution : Type}
    (D : Discourse Person Institution) (inst : Institution) :
    is_reified D inst →
    ∀ (d : Decision Person Institution), d ∈ D → ¬ is_accountable d :=
by
  intro h_reif d hd_in
  have h_inst := h_reif d hd_in
  intro h_account
  obtain ⟨p, hp⟩ := h_account
  rw [hp] at h_inst
  nomatch h_inst

-- Teorema 2: La reificación destruye la accountability
theorem reification_destroys_accountability
    {Person Institution : Type}
    (D : Discourse Person Institution) (inst : Institution) :
    is_reified D inst →
    (∃ d : Decision Person Institution, d ∈ D) →
    ¬ fully_accountable D := by
  intro h_reif ⟨d, hd_in⟩ h_accountable
  have h_not_account := reification_excludes_individual_agency D inst
h_reif d hd_in
  exact h_not_account (h_accountable d hd_in)
```

Explicación conceptual del Teorema 1: 1. Asumimos que el discurso reifica la institución (`h_reif`). 2. Tomamos una decisión `d` que pertenece al discurso. 3. Por definición de reificación, `d.agency = institutional inst`. 4. Asumimos (para reducir al absurdo) que `d` es accountable. 5. Obtenemos la persona concreta `p` responsable. 6. Reescribimos: ahora tenemos `individual p = institutional inst`. 7. Esto es imposible porque los constructores son distintos. Contradicción. ■

Explicación conceptual del Teorema 2: 1. Usamos el Teorema 1 para mostrar que si el discurso reifica y tiene al menos una decisión, esa decisión no es accountable. 2.

Pero `fully_accountable` exige que TODAS las decisiones sean accountable. 3. Contradicción. ■

¿Por qué este teorema importa? Formaliza exactamente lo que afirmamos en el Capítulo 1 sobre el Estado/nación/patria: cuando el discurso dice "el Estado decide", las responsabilidades de los agentes concretos se diluyen. El Teorema 2 demuestra que esa dilución no es accidental: es una **consecuencia lógica necesaria** de la reificación. Si un discurso reifica, entonces por definición no puede ser accountable. No hay forma de tener ambas cosas simultáneamente.

Nota técnica sobre `abbrev`: Usamos `abbrev` en lugar de `def` para definir `Discourse` porque en Lean 4, `def` crea un alias de tipo que no hereda automáticamente las instancias de type class del tipo original. Con `abbrev`, el alias es completamente transparente para el elaborador, lo que permite que la instancia de `Membership` de `List` se resuelva correctamente.

Nota técnica sobre `nomatch`: La táctica `nomatch h_inst` cierra el objetivo porque `h_inst` es una igualdad entre dos constructores distintos de un tipo inductivo (`AgencySource.individual p = AgencySource.institutional inst`). Lean reconoce que no hay forma de que esta igualdad sea verdadera y cierra el objetivo. Si `nomatch` no funciona en alguna versión de Lean, se puede reemplazar por `cases h_inst`.

Nota sobre la naturaleza del teorema: Un lector atento podría señalar que este teorema es, en cierto sentido, una tautología definicional: si definimos la reificación como la exclusión de la agencia individual, y la *accountability* como su requisito, la conclusión se sigue por mera definición. La observación es correcta, pero el valor del teorema no reside en revelar una propiedad oculta del mundo, sino en hacer **explícita y computacionalmente verificable** la consecuencia estructural de nuestras definiciones. Al formalizarlo, demostramos que la dilución de responsabilidad no es un efecto secundario accidental o retórico de la reificación, sino su consecuencia lógica ineludible bajo este modelo.

Asimismo, la exigencia de que la *accountability* recaiga en agentes individuales no es un capricho formal: se ancla en la tradición jurídica y política (desde el derecho romano hasta Kelsen y Hart) que establece que las construcciones institucionales no pueden ser castigadas, encarceladas ni moralmente responsables; en última instancia, solo las personas concretas pueden responder por sus actos.

Qué obtenemos

Con Lean 4 **obtenemos** teoremas verificados. Cuando un teorema compila en Lean 4, **tenemos** la certeza formal de que la conclusión se sigue de las premisas según las reglas de inferencia del sistema. Esto no garantiza que las premisas sean verdaderas (eso es una cuestión empírica o filosófica), pero garantiza que la derivación es válida.

En el contexto de este trabajo, **usamos** Lean 4 para: - Formalizar definiciones precisas de conceptos como "reificación", "estratificación", "competencia normativa". - Demostrar teoremas sobre las propiedades de estos conceptos. - Verificar que ciertas construcciones (como la Aritmética Estratificada de Peano en el Capítulo 3) son consistentes.

Limitaciones

Lean 4 tiene dos limitaciones importantes:

Primera: requiere que el usuario escriba la demostración paso a paso. El sistema verifica, pero no inventa. Si el usuario no sabe cómo demostrar el teorema, Lean 4 no lo ayuda (aunque existen herramientas de automatización que pueden asistir en casos simples).

Segunda: la formalización en Lean 4 es laboriosa. Formalizar una página de matemática estándar puede requerir cientos de líneas de código. Esto significa que no todo lo que afirmamos en este trabajo está formalizado en Lean 4; solo los resultados centrales que justifican el esfuerzo.

2.3 Z3: Solvers SMT y núcleos insatisfacibles

Qué es

Z3 es un solver SMT (Satisfiability Modulo Theories) desarrollado por Microsoft Research.[2] A diferencia de Lean 4, que verifica demostraciones escritas por humanos, **con** Z3 **buscamos** automáticamente asignaciones de valores que satisfagan un conjunto de fórmulas lógicas, o **demostramos** que tales asignaciones no existen.

SMT significa "satisfacibilidad módulo teorías". El solver verifica si un conjunto de fórmulas es satisfacible (existe una asignación de valores que las hace todas verdaderas) dentro de ciertas teorías de fondo (aritmética, arrays, lógica de primer orden, etc.).

Cómo funciona

Un archivo Z3 típico tiene esta estructura:

```
; Declaración de tipos
(declare-sort Agente)
(declare-sort Norma)
(declare-sort Estrato)

; Declaración de predicados
(declare-fun Obligatorio (Agente Norma Estrato) Bool)
(declare-fun Prohibido   (Agente Norma Estrato) Bool)

; Axioma: incompatibilidad deóntica
(assert (forall ((a Agente) (n Norma) (s Estrato))
  (not (and (Obligatorio a n s) (Prohibido a n s)))))

; Consulta: ¿es satisfacible este conjunto de axiomas?
(check-sat)
```

Este código declara: 1. Tres tipos: `Agente`, `Norma`, `Estrato`. 2. Dos predicados: `Obligatorio` y `Prohibido`. 3. Un axioma: no puede ser simultáneamente obligatorio y

prohibido para el mismo agente, norma y estrato. 4. Una consulta: ¿existe una asignación de valores que satisfaga este axioma?

Z3 **devuelve** `sat` (satisfacible) si existe tal asignación, o `unsat` (insatisfacible) si no existe.

La herramienta clave: unsat cores

La funcionalidad más poderosa de Z3 para este trabajo es la generación de **núcleos insatisfacibles** (unsat cores). Cuando un conjunto de fórmulas es insatisfacible, **podemos usar** Z3 para **identificar** el subconjunto mínimo de fórmulas que genera la contradicción.

Por ejemplo, si tenemos diez axiomas y Z3 determina que el conjunto es insatisfacible, **podemos identificar** que solo tres de esos axiomas son responsables de la contradicción. Los otros siete son irrelevantes para el conflicto.

Esto es invaluable para el diagnóstico de errores. En lugar de revisar diez axiomas buscando el problema, **sabemos** exactamente cuáles tres están en conflicto.

Ejemplo concreto: el juez incompetente

En el Capítulo 6 (sistemas normativos), **usaremos** Z3 para demostrar el principio *nemo iudex in causa sua* (nadie puede ser juez en su propia causa). El código tiene esta estructura:

```
(set-option :produce-unsat-cores true)

; Axiomas del sistema
(assert (! (forall ((a Agente) (n Norma))
  (=> (exists ((s0 Estrato)) (Regula n a s0))
      (forall ((s2 Estrato)) (not (Competente a n s2)))))
  :named ax5-incompetencia-reflexiva))

(assert (! (forall ((alfa Acto) (C Contexto))
  (=> (= (Fuerza alfa) 0)
      (= (Valido alfa C)
         (and (CondFeliz alfa C)
              (exists ((s Estrato))
                (Competente (emisor alfa) (norma_aplicable alfa) s))))))
  :named ax3-validez-declarativo))

; Supuesto: un juez intenta juzgarse a sí mismo
(assert (! (= (emisor alfa1) a1) :named teo-juez-emisor))
(assert (! (= (norma_aplicable alfa1) n1) :named teo-juez-norma))
(assert (! (= (Fuerza alfa1) 0) :named teo-juez-fuerza))
(assert (! (Regula n1 a1 s1) :named teo-juez-regula))
(assert (! (CondFeliz alfa1 c1) :named teo-juez-cond))
(assert (! (Valido alfa1 c1) :named teo-juez-querido))
```

```
(check-sat)  ; Devuelve: unsat
(get-unsat-core)
```

Con Z3 **obtenemos** el siguiente resultado:

```
unsat
(ax3-validez-declarativo ax5-incompetencia-reflexiva
 teo-juez-emisor teo-juez-norma teo-juez-querido
 teo-juez-fuerza teo-juez-regula)
```

El núcleo insatisfacible muestra que: - El juez emite un acto válido (`teo-juez-querido`) - Que aplica una norma que lo regula (`teo-juez-regula`) - Pero el axioma de incompetencia reflexiva (`ax5`) dice que si una norma regula a un agente, el agente es incompetente en todos los estratos - Y el axioma de validez (`ax3`) exige que el emisor sea competente en algún estrato

La contradicción es inevitable. **Con** Z3 **la detectamos** y **la aislamos**.

Ejemplo concreto: la inmunidad del crítico

En el mismo script, **formalizamos** el cálculo de reificación y movilización sacrificial (Σ_R). El teorema T1 establece que si un agente cree que una entidad es arbitraria (conoce su carácter construido), entonces no puede sacrificarse por ella:

```
; Teorema T1: Si un agente cree que e es arbitrario, NO se sacrifica por
e.
(push)
(declare-const agT Agente)
(declare-const eT Entidad)
(assert (! (Cree agT (EsArbitrario_prop eT)) :named t1-cree-arbitrario))
(assert (! (Sacrifica agT eT) :named t1-sacrifica))

(check-sat)  ; Devuelve: unsat
(get-unsat-core)
(pop)
```

Z3 **devuelve**:

```
unsat
(sig-A3-reificacion sig-A4-sagrado sig-A6-sacrificio-requiere-sagrado
 t1-sacrifica t1-cree-arbitrario)
```

La cadena lógica es: 1. El agente cree que la entidad es arbitraria (`t1-cree-arbitrario`). 2. Por `sig-A3`, para que la entidad esté reificada, el agente **no debe creer** que es arbitraria. Como sí lo cree, la entidad no está reificada. 3. Por `sig-A4`, lo sagrado equivale a lo reificado. Si no está reificada, no es sagrada. 4. Por `sig-A6`, el sacrificio requiere sacralidad. Si no es sagrada, no puede haber sacrificio. 5. Pero `t1-sacrifica` afirma que hay sacrificio. Contradicción. ■

Con Z3 **demostramos** que la creencia en la arbitrariedad de una entidad y el acto de sacrificio por ella son lógicamente incompatibles. La "ilustración crítica" desactiva la movilización sacrificial.

Qué obtenemos

Con Z3 **obtenemos** certificados de satisfacibilidad o insatisfacibilidad. Cuando Z3 **devuelve** `sat`, **sabemos** que existe un modelo que satisface las fórmulas. Cuando **devuelve** `unsat`, **sabemos** que no existe tal modelo. Y cuando **produce** un unsat core, **identificamos** el subconjunto mínimo responsable de la contradicción.

En el contexto de este trabajo, **usamos** Z3 para: - Verificar la consistencia de conjuntos de axiomas. - Demostrar teoremas por refutación (mostrar que la negación del teorema es insatisfacible). - Diagnosticar contradicciones en sistemas normativos.

Limitaciones

Z3 tiene limitaciones importantes:

Primera: opera en fragmentos decidibles de la lógica. Si las fórmulas caen fuera de estos fragmentos, Z3 puede no terminar o devolver "unknown" (desconocido).

Segunda: Z3 verifica satisfacibilidad, no verdad. Que un conjunto de axiomas sea satisfacible no significa que sea verdadero; significa que es lógicamente posible.

Tercera: Z3 no produce demostraciones legibles por humanos. Produce certificados que pueden ser verificados por otros sistemas, pero no explicaciones en lenguaje natural.

Cuarta (limitación estructural): En SMT-LIB2 (lógica de primer orden), no podemos cuantificar sobre fórmulas ni pasar una fórmula completa como argumento a un predicado como `Cree`. Para modelar creencias, creamos un tipo (o sort, en la terminología técnica de SMT-LIB) no interpretado llamado Prop y funciones que devuelven `Prop` (por ejemplo, `EsArbitrario_prop`). Z3 trata estos términos como tokens opacos: no evalúa su estructura lógica interna ni deduce consecuencias entre ellos. Esto significa que Z3 no modela la **clausura epistémica** (si un agente cree P, y P implica Q, entonces el agente cree Q). Para este trabajo, esa limitación es aceptable porque los axiomas definen las relaciones directamente sobre las creencias atómicas. Pero si en el futuro necesitamos modelar razonamiento deductivo de agentes, deberíamos axiomatizar explícitamente la lógica epistémica.

2.4 Clingo: Razonamiento por defecto y lógica deóntica

Qué es

Clingo es un sistema de programación lógica con Answer Set Programming (ASP) desarrollado por la Universidad de Potsdam.[3] A diferencia de Lean 4 (que verifica demostraciones) y Z3 (que verifica satisfacibilidad), **con** Clingo **generamos** modelos

que satisfacen un conjunto de reglas lógicas, manejando especialmente bien el razonamiento por defecto y la negación por fallo.

Cómo funciona

El siguiente código modela el principio *nemo iudex in causa sua* usando razonamiento por defecto:

```
% Hechos
agente(juan).
agente(maria).
norma(n1).
norma(n2).
estrato(0).
estrato(1).
regula(n1, juan).

% Reglas
competente(A, N, S) :- agente(A), norma(N), estrato(S), not regulado(A,
N).
regulado(A, N) :- regula(N, A).

% Restricción: nadie puede ser competente sobre una norma que lo regula
:- competente(A, N, S), regulado(A, N).

% Consulta
#show competente/3.
```

Este código declara: 1. **Hechos:** Juan y María son agentes. n1 y n2 son normas. Hay dos estratos (0 y 1). La norma n1 regula a Juan. 2. **Reglas:** Un agente es competente sobre una norma en un estrato si no está regulado por esa norma. Un agente está regulado por una norma si la norma lo regula. 3. **Restricción:** No puede ser que un agente sea competente y esté regulado por la misma norma. 4. **Consulta:** Mostrar todas las instancias del predicado `competente/3`.

Resultado verificado

Con Clingo **obtenemos** el siguiente resultado:

```
SATISFIABLE
Answer: 1
competente(juan,n2,0) competente(juan,n2,1)
competente(maria,n1,0) competente(maria,n1,1)
competente(maria,n2,0) competente(maria,n2,1)
```

Interpretación del modelo:

Agente	Norma	¿Competente?	Razón
Juan	n1	**NO** (ausente del modelo)	`regula(n1, juan)` → la condición `not regulado(juan, n1)` falla

Juan	n2	SÍ	n2 no regula a Juan
María	n1	SÍ	n1 no regula a María
María	n2	SÍ	n2 no regula a María

La ausencia de `competente(juan,n1,0)` y `competente(juan,n1,1)` es la demostración del principio *nemo iudex in causa sua*: **Juan no puede ser competente sobre la norma que lo regula**.

Verificación del Grounding Profile: - La regla `competente` tiene **6 instancias** (2 agentes × 2 normas × 2 estratos = 8 posibles, menos 2 por Juan×n1 = 6). ✓ - La restricción `:- competente(A,N,S), regulado(A,N).` tiene **3 matches pero 0 instancias**, lo que significa que fue verificada pero nunca violada. ✓ - `regulado` tiene **1 instancia** (solo `regulado(juan,n1)`). ✓

La herramienta clave: negación por fallo

La funcionalidad distintiva de Clingo es la **negación por fallo** (negation as failure). La expresión `not P` se evalúa como verdadera si no hay evidencia de que `P` sea verdadero. Esto **nos permite** modelar razonamiento por defecto: "asumir P a menos que haya evidencia de lo contrario".

En el contexto de sistemas normativos, esto es crucial. Muchas normas operan por defecto: "los ciudadanos pueden circular libremente a menos que exista una restricción explícita". **Con** Clingo **manejamos** este tipo de razonamiento de manera natural.

Qué obtenemos

Con Clingo **obtenemos** conjuntos de respuestas: modelos que satisfacen el programa lógico. Cada conjunto de respuestas representa una interpretación consistente del programa.

En el contexto de este trabajo, **usamos** Clingo para: - Modelar sistemas normativos con excepciones y razonamiento por defecto. - Verificar la consistencia de conjuntos de normas. - Generar escenarios que satisfacen ciertas condiciones.

Limitaciones

Clingo tiene limitaciones:

Primera: la semántica de ASP es diferente de la lógica clásica. Los conjuntos de respuestas no son modelos en el sentido de la teoría de modelos estándar.

Segunda: Clingo puede generar múltiples conjuntos de respuestas, y el usuario debe interpretar cuál es relevante para su análisis.

Tercera: ASP no maneja bien la cuantificación sobre fórmulas o la lógica de orden superior.

2.5 Matriz problema-herramienta

El Capítulo 0 estableció una matriz que asocia tipos de problemas con herramientas específicas. La reproducimos aquí con explicaciones:

Tipo de problema	Herramienta	Justificación
Consistencia lógica, teoremas con inducción	Lean 4	Kernel pequeño, prueba verificable, lógica de primer orden con igualdad
Satisfactibilidad de conjuntos de axiomas en fragmentos decidibles	Z3 / SMT-LIB	Automático, produce modelos o unsat core
Normas con excepciones, razonamiento por defecto	Clingo / Deolingo	Answer Set Programming maneja negación por fallo y reglas defeasibles
Verificación de tipos, estructura de datos	Lean 4	Teoría de tipos dependientes, chequeo en compilación

La elección de herramienta se justifica en cada capítulo. No utilizamos una herramienta por inercia, sino porque es la adecuada para el problema concreto.

2.6 Cómo leer el código en este trabajo

Los capítulos siguientes (especialmente el 3, 5 y 6) incluyen fragmentos de código en Lean 4, Z3 y Clingo. El lector puede abordarlos a dos niveles:

Nivel conceptual: Ignorar el código y leer solo la prosa que lo explica. La prosa describe qué **hacemos con** el código, qué teorema **demostramos**, y qué implicaciones **tiene** para el argumento. Este nivel es suficiente para entender el argumento general.

Nivel técnico: Leer el código y verificar que efectivamente **hace** lo que la prosa afirma. Este nivel requiere familiaridad con los lenguajes, pero el código está disponible para quien quiera inspeccionarlo.

Para facilitar la lectura conceptual, cada fragmento de código va acompañado de: 1. **Descripción en prosa** de qué **hacemos con** el código. 2. **Explicación línea por línea** de las partes clave. 3. **Interpretación** de qué significa el resultado para el argumento.

El código no es decorativo. Cada fragmento que incluimos compila y produce el resultado que afirmamos. Ningún fragmento se incluye sin que haya sido verificado por nosotros.

2.7 El estatus filosófico de los axiomas: elecciones, no descubrimientos

Toda formalización matemática o lógica comienza con un acto fundacional que las herramientas computacionales no pueden auditar: la elección de los **axiomas**. Para comprender el alcance y los límites de lo que hacemos en este trabajo con Lean 4, Z3 y

Clingo, es imperativo desarmar la anatomía de una demostración formal y ser radicalmente transparentes sobre el estatus filosófico de sus cimientos.

La anatomía de una demostración formal

En cualquier sistema formal, el conocimiento se estructura en una cadena de dependencias lógicas: 1. **Axiomas:** Son las premisas fundacionales, las reglas del juego que nosotros, los autores, decidimos adoptar sin demostrar dentro del sistema. 2. **Lemas:** Son teoremas auxiliares, pasos intermedios que demostramos para facilitar la prueba de resultados mayores. 3. **Teoremas:** Son las proposiciones sustantivas que nos interesa demostrar, derivadas lógicamente de los axiomas y lemas. 4. **Corolarios:** Son consecuencias inmediatas y directas de los teoremas.

Las herramientas formales que presentamos en este capítulo (los *kernels* de Lean, los motores de Z3) son auditoras infalibles de los eslabones de esta cadena. Si nosotros afirmamos que un teorema se sigue de un lema, y el lema se sigue de un axioma, el software verifica que la deducción sea estructuralmente impecable. **Pero las herramientas son ciegas a los axiomas.** Ningún compilador puede verificar si un axioma es "verdadero" en el mundo empírico o "justo" en el mundo social; solo puede verificar que, *si* el axioma se asume como verdadero, las consecuencias se siguen necesariamente.

El riesgo de reificar los axiomas

Aquí es donde el método de este libro enfrenta su prueba de fuego más delicada. Si nosotros presentamos nuestros axiomas como "descubrimientos empíricos" o como "leyes naturales del comportamiento social", estaríamos cometiendo exactamente el mismo error de reificación que denunciamos en el Capítulo 1: estaríamos tratando una construcción discursiva (nuestra elección de premisas) como si fuera una sustancia con existencia autónoma.

La formalización no elimina la interpretación filosófica; la desplaza al nivel de los axiomas. Por lo tanto, para evitar la falacia de la herramienta (creer que lo formalizado es objetivamente verdadero), nosotros debemos declarar explícitamente que **nuestros axiomas no son descubrimientos, sino elecciones filosóficas**. Son compromisos que adoptamos porque resultan fecundos para el proyecto de análisis que estamos realizando, y porque se anclan en tradiciones de pensamiento específicas.

El anclaje filosófico de nuestras elecciones

Para que el lector pueda evaluar críticamente nuestro trabajo, debemos transparentar el anclaje filosófico de los axiomas que utilizaremos en los capítulos subsiguientes. A modo de ejemplo:

- **Sobre la agencia y la *accountability* (Capítulo 2 / Lean 4):** Nosotros definimos un tipo `AgencySource` con exactamente dos constructores: `individual` (personas físicas) e `institutional` (construcciones). ¿Por qué esta elección y no otra? Esta decisión no es una verdad universal; se ancla en la **tradición jurídica occidental**

(desde el derecho romano hasta Kelsen y Hart), que establece que las construcciones institucionales pueden ser sancionadas o disueltas, pero solo las personas de carne y hueso pueden ser penalmente responsables o moralmente *accountable*. Si el lector prefiriera una ontología que incluya "agencia colectiva difusa" o "agencia ecológica", nuestros teoremas sobre la dilución de responsabilidad cambiarían.

- **Sobre la jerarquía normativa (Capítulo 6 / Z3):** Nosotros axiomatizamos que si una norma regula a un agente, ese agente es incompetente para evaluar esa misma norma (el principio *nemo iudex in causa sua*). Esta elección se ancla en la **Teoría Pura del Derecho de Kelsen** y en la lógica deóntica, que exigen una estratificación jerárquica para evitar las paradojas de la autoevaluación.
- **Sobre la reificación y el sacrificio (Capítulo 6 / Z3):** Nosotros axiomatizamos que un agente no puede sacrificarse por una entidad si es plenamente consciente de su arbitrariedad constitutiva. Esta elección se ancla en la **sociología crítica de Pierre Bourdieu** y en la teoría de la violencia simbólica, que postulan que la dominación requiere el desconocimiento (la *misperception*) de su propia contingencia por parte del dominado.

La naturaleza condicional de los teoremas

Como consecuencia directa de lo anterior, todos los teoremas que demostramos en este trabajo son **estrictamente condicionales**.

Cuando nosotros demostramos que "la reificación institucional destruye la *accountability*" o que "la creencia en la arbitrariedad bloquea el sacrificio", no estamos dictando leyes inmutables de la sociología. Estamos afirmando lo siguiente: *"Bajo los axiomas que nosotros hemos elegido, anclados en las tradiciones jurídica y sociológica mencionadas, estas propiedades se siguen necesariamente"*.

El lector tiene absoluta libertad para cuestionar nuestros axiomas, rechazarlos y proponer otros. Si lo hace, las herramientas formales seguirán a su disposición para que verifique las nuevas consecuencias lógicas de sus propias premisas. Esta invitación a la refutación no es una debilidad de nuestro método; es su mayor fortaleza. Es la garantía de que estamos utilizando la formalización como una herramienta de higiene conceptual transparente, y no como un manto de autoridad matemática para ocultar decisiones ideológicas o filosóficas bajo la alfombra de la sintaxis.

2.8 Limitaciones y honestidad metodológica

Las herramientas formales son poderosas, pero no son mágicas. Tienen limitaciones que es importante reconocer:

Primera limitación: las herramientas verifican derivaciones, no verdad. Que un teorema compile en Lean 4 garantiza que la conclusión se sigue de las premisas, pero no garantiza que las premisas sean verdaderas. La verdad de las premisas es una cuestión empírica o filosófica que las herramientas no pueden resolver.

Segunda limitación: la formalización requiere elecciones. Formalizar un concepto en lenguaje natural requiere decidir cómo representarlo en el lenguaje formal. Estas decisiones no son neutrales; pueden introducir sesgos o perder matices. Por ejemplo, formalizar "la patria exige sacrificio" como `Exige(Patria, Sacrificio)` pierde la ambigüedad del lenguaje natural, pero esa pérdida es el precio de la precisión.

Tercera limitación: no todo es formalizable. Hay aspectos del análisis de la reificación que no pueden formalizarse completamente. La interpretación de los efectos de poder, la evaluación de la relevancia histórica, la crítica ideológica: estos son juicios que operan en lenguaje natural y no pueden reducirse a código. Las herramientas formales complementan el análisis, no lo reemplazan.

Cuarta limitación: las herramientas tienen bugs. Aunque el kernel de Lean 4 es extremadamente pequeño y ha sido revisado exhaustivamente, el sistema completo (bibliotecas, tácticas, interfaz) tiene bugs. Z3 y Clingo también tienen bugs conocidos. Confiar ciegamente en las herramientas sería ingenuo. Por eso verificamos los resultados con múltiples métodos cuando es posible.

Quinta limitación: la formalización no elimina la reificación, la desplaza. Las herramientas formales también tienen sintaxis, y sus tipos y términos pueden cosificarse conceptualmente. La diferencia es que en un sistema formal la reificación está controlada por las reglas de inferencia, pero el significado último de los símbolos sigue dependiendo de la interpretación humana. Lo formal no escapa completamente de la reificación; la hace explícita y controlable.

Advertencia metodológica: Incluso al hablar de herramientas formales, el lenguaje natural nos obliga a usar formulaciones que atribuyen agencia a las herramientas ("Lean verifica", "el kernel acepta", "Z3 demuestra"). Esto es inevitable. Lo importante es recordar que estas formulaciones son abreviaturas de operaciones más complejas donde los agentes reales somos nosotros: nosotros escribimos el código, nosotros interpretamos los resultados, nosotros decidimos qué formalizar.

El estatus epistémico de los resultados formales

Siguiendo el sistema de etiquetado establecido en el Capítulo 0, los resultados formales se clasifican como **[A] Probada por código**. Esto significa que el resultado ha sido verificado por una herramienta formal y compila sin errores.

Pero es crucial entender qué significa esto. Un resultado [A] no es "más verdadero" que un resultado [B] (citado de literatura) o [C] (interpretación). Significa que tiene un tipo diferente de respaldo: un respaldo computacional en lugar de empírico o hermenéutico.

Un teorema verificado en Lean 4 garantiza que, bajo ciertos axiomas y reglas de inferencia, la conclusión se sigue necesariamente. Pero los axiomas mismos son elecciones que requieren justificación filosófica. La herramienta verifica la derivación; nosotros justificamos las premisas.

2.9 Cierre: las herramientas como prótesis cognitiva

Las herramientas formales son, en última instancia, **prótesis cognitivas**. Así como **con** un telescopio **extendemos** nuestra capacidad para ver objetos lejanos, **con** las herramientas formales **extendemos** nuestra capacidad para detectar inconsistencias, verificar derivaciones y diagnosticar errores.

Pero una prótesis no reemplaza al órgano; lo complementa. **Con** el telescopio **no vemos** por sí mismo; **requerimos** interpretar lo que **observamos**. Del mismo modo, **con** las herramientas formales **no razonamos** por sí mismas; **requerimos** formular las preguntas, interpretar los resultados y integrarlos en un argumento más amplio.

Nosotros usamos herramientas formales porque el análisis de la reificación requiere precisión que el lenguaje natural no puede proporcionar por sí solo. Pero las herramientas no son este trabajo. Este trabajo es el argumento conceptual que **construimos con ayuda de** las herramientas. El lector que ignore el código perderá detalles técnicos, pero no perderá el argumento central. El lector que estudie el código ganará certeza sobre los resultados formales, pero seguirá necesitando interpretar su significado.

Las herramientas formales son el microscopio. Lo que **observamos** a través de él es lo que **mostramos** en este trabajo.

Resumen de códigos verificados en este capítulo

Herramienta	Código	Resultado	Verificado
Lean 4	`reification_is_category_mistake`	Compila sin errores	✓
Lean 4	`reification_excludes_individual_agency`	Compila sin errores	✓
Lean 4	`reification_destroys_accountability`	Compila sin errores	✓
Z3	SFCD (consistencia)	`sat`	✓
Z3	Juez incompetente	`unsat` + unsat core	✓
Z3	Σ_R (consistencia)	`sat`	✓
Z3	T1 (inmunidad del crítico)	`unsat` + unsat core	✓
Clingo	Nemo iudex (competencia normativa)	`SATISFIABLE`, 6 instancias	✓

Capítulo 3: SPA — Un sistema aritmético estratificado

3.0 Una puerta de entrada: la paradoja del mentiroso

Antes de presentar el sistema formal, conviene detenerse en el problema que lo motiva. La paradoja del mentiroso es una de las más antiguas de la tradición lógica: un enunciado que afirma "esta oración es falsa". Si es verdadera, entonces es falsa. Si es falsa, entonces es verdadera. La paradoja no se resuelve con más información sobre el mundo; se resuelve con una restricción sobre la estructura del lenguaje que la formula.

En 1933, Alfred Tarski demostró que la paradoja del mentiroso no es un juego de palabras: tiene una estructura formal precisa que puede reproducirse en cualquier lenguaje lo suficientemente expresivo como para contener su propio predicado de verdad [B]. La consecuencia es un teorema de imposibilidad: ningún lenguaje formal puede contener su propia definición de verdad sin volverse inconsistente.

¿Por qué esto importa para el proyecto de este trabajo? Porque la paradoja de Tarski es, en su estructura más profunda, un caso de reificación: el predicado de verdad —que es una fórmula, una relación sintáctica entre un código y una propiedad— se trata como si fuera una sustancia autorreferente, una entidad capaz de aplicarse a sí misma. Es exactamente el tipo de error categorial que el Capítulo 1 identifica: una construcción de tipo "relación" es tratada como si tuviera propiedades de tipo "sustancia". La estratificación impide este error de tipo mediante una restricción sintáctica explícita.

En este capítulo presentamos SPA (Stratified Peano Arithmetic), un sistema formal que nosotros diseñamos y formalizamos en Lean 4 para implementar la solución de Tarski de manera verificada mecánicamente. SPA no es una invención teórica nueva: la solución que implementa es conocida desde hace décadas. Lo que SPA aporta es una implementación completa, estratificada, y verificada sin omisiones.

3.1 El problema que SPA resuelve: el teorema de Tarski [B]

El teorema de Tarski establece la indefinibilidad de la verdad: ninguna fórmula del lenguaje de la aritmética puede definir el conjunto de sentencias verdaderas de ese mismo lenguaje [B].

La demostración es una variante formal de la paradoja del mentiroso. Si existiera una fórmula $V(x)$ que definiera la verdad —es decir, $V(\ulcorner \varphi \urcorner)$ fuera equivalente a "φ es verdadera"—, entonces por diagonalización podríamos construir una oración λ que afirmara "yo soy falsa":

$$\lambda \equiv \neg V(\ulcorner \lambda \urcorner)$$

Si λ es verdadera, entonces $V(\ulcorner \lambda \urcorner)$ es falsa, y por tanto λ es falsa. Si λ es falsa, entonces $V(\ulcorner \lambda \urcorner)$ es verdadera, y por tanto λ es verdadera. Contradicción.

El problema no reside en la noción de verdad en sí, sino en pretender que el predicado de verdad habite el mismo nivel lógico que las fórmulas que evalúa. Cuando el lenguaje y el metalenguaje se confunden, la autorreferencia destructiva es inevitable.

Estado del problema. Este no es un problema abierto. Tarski mismo propuso una solución: la jerarquía de metalenguajes [B]. La teoría de tipos de Russell (1908) ya había introducido una distinción similar para las paradojas conjuntistas [B]. Existen otras formalizaciones del teorema de completitud en asistentes de pruebas (Isabelle/HOL, Coq), pero SPA se distingue por hacer de la estratificación un mecanismo sintáctico explícito, con operadores intensionales (`provable_op`, `true_op`) cuyo tipado refleja la jerarquía de niveles y cuya corrección está garantizada por el verificador de tipos de Lean [A].

3.2 Aplicación del protocolo al caso de Tarski [C]

Como mostramos en 3.0, la paradoja de Tarski es un caso de reificación: el predicado de verdad (una relación sintáctica) se trata como si fuera una sustancia autorreferente. Apliquemos ahora las cuatro preguntas del Capítulo 0 a este caso:

Pregunta 1: ¿Cuál es el objeto formal y cuál es su tipo?

El predicado de verdad $V(x)$ es una fórmula de la aritmética de primer orden. Su tipo nativo es `ArithForm s` para algún nivel s. Su argumento x es un numeral que codifica una fórmula del mismo nivel s.

Pregunta 2: ¿Qué atribuye la narrativa a este objeto que su tipo no contiene?

La definición informal de $V(x)$ presupone que puede aplicarse a todas las fórmulas del lenguaje, incluidas aquellas que contienen al propio V. Es decir, presupone que V pertenece al mismo nivel que las fórmulas que evalúa. Esta presuposición no está forzada por el tipo de V; es una decisión de diseño implícita en la formulación estándar.

Pregunta 3: ¿Existe una descripción alternativa sin ese atributo?

Sí. La jerarquía de metalenguajes de Tarski —y, de manera más general, la teoría de tipos— muestra que la verdad puede definirse de manera consistente si se respeta una distinción de niveles: el predicado de verdad para fórmulas del nivel n debe residir en el nivel $n + 1$. SPA no inventa esta solución, pero la implementa de manera completa y verificada [A].

Pregunta 4: ¿Es teoría resuelta con narrativa sucia?

Sí. Tarski resolvió el problema técnico en 1933 [B]. La jerarquía de lenguajes es una solución conocida desde hace décadas. El desafío no es encontrar una solución, sino formalizarla completamente y mostrar que no sacrifica la completitud del sistema.

3.3 La solución de SPA: operadores estratificados [A/B]

La solución que nosotros implementamos en SPA es directa: los operadores que evalúan fórmulas deben residir en un nivel superior a las fórmulas evaluadas. En la sintaxis de SPA, esto se implementa mediante constructores que incluyen un testigo explícito de la relación de niveles:

```
ArithForm.provable_op : {s_lower : Nat} → (s = s_lower + 1) → ArithTerm s
→ ArithForm s
ArithForm.true_op     : {s_lower : Nat} → (s = s_lower + 1) → ArithTerm s
→ ArithForm s
```

El parámetro `h : s = s_lower + 1` es un testigo que certifica que el nivel s es estrictamente superior al nivel s_{lower} de la fórmula evaluada. Sin este testigo, la fórmula no está bien formada. El verificador de tipos de Lean rechaza cualquier intento de construir `provable_op` o `true_op` sin esta certificación.

¿Qué representa el argumento de tipo `ArithTerm s`? No es la fórmula misma que se evalúa, sino un término que codifica esa fórmula. La función de serialización $\ulcorner \cdot \urcorner$: ArithForm $s_{\text{lower}} \to$ ArithTerm s convierte una fórmula del nivel inferior en un numeral que la representa. Así, `true_op h` ($\ulcorner \phi \urcorner$) es una fórmula del nivel s que afirma "la fórmula φ del nivel s_{lower} es verdadera".

En términos de tipos, esto significa que una fórmula en el nivel n puede contener `provable_op` o `true_op` aplicados a términos que codifican fórmulas del nivel $n-1$. No existe ninguna manera de construir `provable_op` o `true_op` aplicados a fórmulas del mismo nivel.

Nota importante. Esto no elimina toda forma de autorreferencia. La diagonalización sigue siendo posible dentro de cada nivel: en el nivel n puede construirse una fórmula que hable de la demostrabilidad de fórmulas del nivel $n-1$. Lo que se bloquea es la autorreferencia destructiva: aquella que requiere que el predicado de verdad o demostrabilidad pertenezca al mismo nivel que la fórmula evaluada. La autorreferencia se reorganiza jerárquicamente, no se suprime.

Diagrama de la jerarquía de niveles

Para visualizar cómo opera la estratificación, consideremos los tres primeros niveles:

```
Nivel 2: fórmulas que pueden hablar de verdad/demostrabilidad del nivel 1
    ↑ true_op, provable_op (con testigo s = s_lower + 1)
Nivel 1: fórmulas que pueden hablar de verdad/demostrabilidad del nivel 0
    ↑ true_op, provable_op (con testigo s = s_lower + 1)
Nivel 0: fórmulas aritméticas básicas (sin operadores de verdad)
```

Cada nivel puede evaluar fórmulas del nivel inmediatamente inferior, pero no puede evaluarse a sí mismo. Esto impide la formación de la oración paradójica λ que afirma "yo soy falsa", porque tal oración requeriría que el predicado de verdad habitara el mismo nivel que la fórmula que evalúa.

Ejemplo concreto de una fórmula bien formada

Para ilustrar cómo funciona la estratificación sintáctica, consideremos un ejemplo simple. Supongamos que queremos expresar en el nivel 1 que "la fórmula $0 = 0$ del nivel 0 es verdadera":

```
-- La fórmula "0 = 0" del nivel 0:
def zero_eq_zero : ArithForm 0 :=
  .eq (.zero : ArithTerm 0) (.zero : ArithTerm 0)
-- Su código de serialización (un numeral):
def code_zero_eq_zero : ArithTerm 1 :=
  serialize zero_eq_zero  -- función que codifica la fórmula como numeral
-- La afirmación "0 = 0 es verdadera" en el nivel 1:
def truth_claim : ArithForm 1 :=
  .true_op (s := 1) (s_lower := 0) (by simp) code_zero_eq_zero
```

El testigo `(by simp)` certifica que $1 = 0 + 1$. Si intentáramos construir `.true_op` en el nivel 0 aplicado a una fórmula del nivel 0, Lean rechazaría la construcción en tiempo de compilación con un error de tipo. La estratificación no es una convención que nosotros respetemos voluntariamente; es una restricción impuesta por el verificador de tipos.

3.4 Sintaxis formal de SPA [A]

La sintaxis de SPA se define mediante tipos inductivos en Lean 4, indexados por un parámetro de nivel $s : \mathbb{N}$. Presentamos las definiciones y luego las explicamos paso a paso.

Términos aritméticos (`ArithTerm s`):

```
inductive ArithTerm (s : Nat) : Type where
  | var    : Nat → ArithTerm s
  | const  : NatS s → ArithTerm s
  | henkin : HenkinConst → ArithTerm s
  | zero   : ArithTerm s
  | succ   : ArithTerm s → ArithTerm s
  | add    : ArithTerm s → ArithTerm s → ArithTerm s
  | mul    : ArithTerm s → ArithTerm s → ArithTerm s
  | exp    : ArithTerm s → ArithTerm s → ArithTerm s
  | pred   : ArithTerm s → ArithTerm s
```

Explicación para el lector no especializado. Los términos son las "expresiones numéricas" del sistema: variables, constantes, cero, sucesor, suma, multiplicación, exponenciación y predecesor. El parámetro `s` no afecta la estructura interna de los términos; actúa como una "etiqueta de nivel" que impide mezclar términos de niveles distintos en fórmulas estratificadas. Las constantes de Henkin (`henkin`) se incluyen para la construcción posterior de testigos existenciales.

Fórmulas aritméticas (`ArithForm s`):

```
inductive ArithForm (s : Nat) : Type where
  | bottom      : ArithForm s
  | eq          : ArithTerm s → ArithTerm s → ArithForm s
  | impl        : ArithForm s → ArithForm s → ArithForm s
  | neg         : ArithForm s → ArithForm s
  | forall_form : Nat → ArithForm s → ArithForm s
  | provable_op : {s_lower : Nat} → (s = s_lower + 1) → ArithTerm s →
ArithForm s
  | true_op     : {s_lower : Nat} → (s = s_lower + 1) → ArithTerm s →
ArithForm s
  | list_nth_eq : ArithTerm s → ArithTerm s → ArithTerm s → ArithForm s
```

Explicación para el lector no especializado. Las fórmulas son las "oraciones" del sistema: absurdo (`bottom`), igualdad (`eq`), implicación (`impl`), negación (`neg`), cuantificación universal (`forall_form`), y los dos operadores estratificados (`provable_op` y `true_op`). El constructor `list_nth_eq` es un auxiliar técnico para la serialización de listas.

Nota sobre la compilación. Los fragmentos de código que aparecen en las secciones 3.4 a 3.7 son extractos ilustrativos de un sistema más grande (SPA, 27.600 líneas de código Lean 4). Estos fragmentos no compilarán de forma aislada porque dependen de definiciones auxiliares (`NatS`, `HenkinConst`, `serialize`, etc.) que no están incluidas aquí por razones de espacio. Para verificar la compilación completa de SPA, es necesario acceder al repositorio del proyecto [enlace]. Los fragmentos se incluyen para ilustrar la estructura del sistema, no para ser ejecutados directamente.

3.4.1 Metateoremas de estratificación sintáctica [A-Meta]

Los siguientes metateoremas se derivan directamente de la sintaxis estratificada de SPA. No introducen nuevo código Lean: son demostraciones formales en prosa [A-Meta] que explicitan consecuencias estructurales de las firmas de tipo definidas en la sección 3.4.

Metateorema T-NoInt-0 [A-Meta]

El nivel 0 no admite operadores intensionales.

Enunciado. Ninguna fórmula `ϕ : ArithForm 0` contiene subfórmulas de la forma `true_op h t` ni `provable_op h t`.

Demostración. Procedemos por inducción estructural sobre `ϕ : ArithForm 0`.

Casos base: Si `ϕ = bottom` o `ϕ = eq t₁ t₂`, no contiene ningún constructor intensional por definición.

Paso inductivo: Supongamos que ϕ se construye mediante `impl`, `neg` o `forall_form` a partir de fórmulas que no contienen operadores intensionales. Entonces ϕ tampoco los contiene, pues estos constructores no introducen `true_op` ni `provable_op`.

Finalmente, supongamos que `ϕ = true_op h t` o `ϕ = provable_op h t`. Por la firma de tipo de ambos constructores, se requiere un testigo `h : s = s_lower + 1` donde `s` es el nivel de `ϕ`. Como `s = 0`, tendríamos `0 = s_lower + 1` para algún `s_lower ∈ ℕ`, lo cual es aritméticamente imposible. Por tanto, estos constructores no pueden aparecer en el nivel 0. ■

Metateorema T-Mono [A-Meta]

Monotonicidad estricta de la estratificación.

Enunciado. Si `ϕ : ArithForm s` contiene una subfórmula `true_op h t` o `provable_op h t`, entonces `h` es un testigo de `s = s_lower + 1` para algún `s_lower < s`.

Demostración. Sea `ϕ : ArithForm s` que contiene `true_op h t`. Por la firma de tipo del constructor

```
true_op : {s_lower : Nat} → (s = s_lower + 1) → ArithTerm s → ArithForm s
```

el argumento `h` tiene tipo `s = s_lower + 1`. Como `s_lower ∈ ℕ`, se sigue `s = s_lower + 1 ≥ 1` y por tanto `s_lower = s - 1 < s`. El caso de `provable_op` es idéntico por su firma análoga. ■

Metateorema T-NoLambda [A-Meta]

Imposibilidad de la oración paradójica de Tarski.

Enunciado. No existe ninguna fórmula `λ : ArithForm s` tal que `λ = true_op h (⌜ λ ⌝)` para algún testigo `h`.

Demostración. Supongamos, para reducir al absurdo, que existe tal `λ`. Entonces `λ : ArithForm s` y, por la definición de `true_op`, se requiere un testigo `h : s = s_lower + 1` tal que el término `⌜ λ ⌝ : ArithTerm s` codifique una fórmula del nivel `s_lower`.

Pero `⌜ λ ⌝` es la serialización de `λ` misma, que reside en el nivel `s`. Luego la fórmula codificada por `⌜ λ ⌝` está en nivel `s`, no en nivel `s_lower`. Para que `true_op h (⌜ λ ⌝)` esté bien formado, el término debe codificar una fórmula de nivel `s_lower`, lo que fuerza `s = s_lower`. Pero el testigo `h` afirma `s = s_lower + 1`. Entonces `s = s + 1`, contradicción aritmética en `ℕ`. ■

Metateorema T-NoCycle [A-Meta]

Imposibilidad de ciclos de referencia intensional.

Enunciado. No existe una secuencia finita de fórmulas $\phi_1, \ldots, \phi_n$ con $n \geq 1$ tal que, para cada $i \in \{1, \ldots, n\}$, ϕ_i contenga `true_op` h_i $(\ulcorner \phi_{i+1} \urcorner)$ (con índices módulo n).

Demostración. Supongamos que existe tal ciclo. Sea s_i el nivel de ϕ_i. Por T-Mono, cada ocurrencia de `true_op` en ϕ_i referenciando ϕ_{i+1} impone $s_i = s_{i+1} + 1$. Recorriendo el ciclo:

$$s_1 = s_2 + 1 = s_3 + 2 = \cdots = s_n + (n-1) = s_1 + n.$$

Luego $s_1 = s_1 + n$, lo cual implica $n = 0$. Pero la hipótesis exige $n \geq 1$. Contradicción. ■

Metateorema T-HomoEq [A-Meta]

Homogeneidad de nivel en la igualdad.

Enunciado. Si $\mathtt{eq}\ t_1\ t_2 : \mathtt{ArithForm}\ s$, entonces $t_1 : \mathtt{ArithTerm}\ s$ y $t_2 : \mathtt{ArithTerm}\ s$.

Demostración. Directo de la firma de tipo del constructor

$\mathtt{eq} : \mathtt{ArithTerm}\ s \to \mathtt{ArithTerm}\ s \to \mathtt{ArithForm}\ s$

Ambos argumentos deben pertenecer al mismo tipo $\mathtt{ArithTerm}\ s$. ■

Metateorema T-NoMix [A-Meta]

Imposibilidad de mezclar niveles en fórmulas atómicas.

Enunciado. No existen $s_1, s_2 \in \mathbb{N}$ con $s_1 \neq s_2$, $t_1 : \mathtt{ArithTerm}\ s_1$ y $t_2 : \mathtt{ArithTerm}\ s_2$ tales que $\mathtt{eq}\ t_1\ t_2$ esté bien formado.

Demostración. Supongamos, para reducir al absurdo, que $\mathtt{eq}\ t_1\ t_2$ está bien formado con $t_1 : \mathtt{ArithTerm}\ s_1$ y $t_2 : \mathtt{ArithTerm}\ s_2$. Por la firma de tipo de eq, ambos argumentos deben pertenecer a $\mathtt{ArithTerm}\ s$ para algún s común. Luego $\mathtt{ArithTerm}\ s_1 = \mathtt{ArithTerm}\ s$, lo que fuerza $s_1 = s$ por inyectividad del índice de tipo, y análogamente $s_2 = s$. Por tanto $s_1 = s_2$, contradiciendo la hipótesis. ■

Metateorema T-IndStruct [A-Meta]

Principio de inducción estructural para fórmulas estratificadas.

Enunciado. Sea $s \in \mathbb{N}$ fijo y sea $P : \mathtt{ArithForm}\ s \to \mathtt{Prop}$ una propiedad. Si:

- $P(\mathtt{bottom})$;
- $\forall\ t_1\ t_2 : \mathtt{ArithTerm}\ s,\ P(\mathtt{eq}\ t_1\ t_2)$;
- $\forall\ \phi\ \psi : \mathtt{ArithForm}\ s,\ P(\phi) \wedge P(\psi) \to P(\mathtt{impl}\ \phi\ \psi)$;
- $\forall\ \phi : \mathtt{ArithForm}\ s,\ P(\phi) \to P(\mathtt{neg}\ \phi)$;
- $\forall\ x : \mathbb{N},\ \forall\ \phi : \mathtt{ArithForm}\ s,\ P(\phi) \to P(\mathtt{forall_form}\ x\ \phi)$;
- $\forall\ \mathtt{s_lower} : \mathbb{N},\ \forall\ h : s = \mathtt{s_lower} + 1,\ \forall\ t : \mathtt{ArithTerm}\ s,\ P(\mathtt{true_op}\ h\ t)$;
- $\forall\ \mathtt{s_lower} : \mathbb{N},\ \forall\ h : s = \mathtt{s_lower} + 1,\ \forall\ t : \mathtt{ArithTerm}\ s,\ P(\mathtt{provable_op}\ h\ t)$;

entonces $\forall\ \phi : \mathtt{ArithForm}\ s,\ P(\phi)$.

Demostración. $\mathtt{ArithForm}\ s$ es un tipo inductivo con siete constructores. Por el principio general de inducción para tipos inductivos (disponible en la lógica constructiva subyacente a Lean), toda propiedad que se cumple en los constructores base y se preserva por los constructores inductivos se extiende a todos los habitantes

del tipo. Las siete cláusulas enumeradas corresponden exactamente a los siete constructores de `ArithForm s`. ■

3.5 Sistema deductivo [A]

El sistema deductivo de SPA es una lógica de primer orden con igualdad, axiomas de Peano, y un esquema de inducción. Se define como un predicado inductivo en Lean:

```
inductive SPA_Deduction_Internal (s : Nat) : Set (ArithForm s) → ArithForm s → Prop
```

Las reglas incluyen:

Lógica proposicional: introducción y eliminación de implicación, doble negación, reducción al absurdo.

Cuantificación: introducción y eliminación del cuantificador universal, con condiciones de frescura (`NotFreeInSet`, `IsFreeFor`) para evitar captura de variables.

Igualdad: reflexividad, simetría, transitividad, y reglas de congruencia para todos los constructores de términos y fórmulas.

Axiomas de Peano: $\neg(S(x) = 0)$ y $S(x) = S(y) \to x = y$.

Inducción: el esquema estándar para fórmulas de primer orden.

Nota sobre los axiomas. Siguiendo la reflexión que desarrollamos en la sección 2.7 del Capítulo 2, nosotros reconocemos que estos axiomas son elecciones filosóficas, no descubrimientos empíricos. Los axiomas de Peano se anclan en la tradición matemática estándar. La lógica de primer orden con igualdad se ancla en la tradición analítica. La inducción se ancla en la aritmética finitaria.

¿Por qué PA y no otra aritmética? Elegimos la Aritmética de Peano (PA) porque es la elección estándar para formalizaciones de aritmética de primer orden: es suficientemente expresiva para codificar la sintaxis (necesario para el teorema de Tarski), es recursivamente axiomatizable (necesario para aplicar los teoremas de Gödel), y es la base estándar en la literatura de verificación formal (Mathlib, etc.). Aritméticas más débiles como Robinson Q no tienen suficiente potencia inductiva para la construcción de Henkin-Lindenbaum. Aritméticas intuicionistas requerirían un tratamiento completamente diferente de la lógica subyacente. Nosotros elegimos PA porque es el marco estándar donde la pregunta de Tarski se formula tradicionalmente [B].

Nosotros elegimos estos axiomas porque son fecundos para el proyecto de formalización estratificada, y porque permiten demostrar la completitud sin sacrificar la consistencia. El lector puede cuestionar estos axiomas y proponer otros; en ese caso, los teoremas pueden cambiar.

3.5.1 Metateoremas de cierre y preservación [A-Meta]

Los siguientes metateoremas explicitan propiedades de preservación de nivel asociadas al sistema deductivo y a las operaciones sintácticas auxiliares. Son demostraciones formales en prosa [A-Meta] y no introducen nuevo código.

Metateorema T-Ded-Level [A-Meta]

El sistema deductivo es cerrado por nivel.

Enunciado. Si `SPA_Deduction_Internal s Γ ϕ`, entonces `Γ ⊆ ArithForm s` y `ϕ : ArithForm s`.

Demostración. El predicado inductivo `SPA_Deduction_Internal` tiene tipo

```
Nat → Set (ArithForm s) → ArithForm s → Prop
```

Por definición, el parámetro de nivel `s` indexa tanto el conjunto de premisas como la conclusión. Cada regla de inferencia del sistema deductivo (introducción y eliminación de implicación, cuantificación, igualdad, axiomas de Peano, inducción) opera exclusivamente sobre fórmulas de `ArithForm s`. Luego, por inducción sobre la derivación, todas las fórmulas que aparecen en la prueba pertenecen a `ArithForm s`. ■

Metateorema T-Subst-Level [A-Meta]

La sustitución preserva el nivel.

Enunciado. Sea `ϕ : ArithForm s` y sea σ una sustitución que asigna a cada variable un término de `ArithTerm s`. Entonces la sustitución `ϕ[σ] : ArithForm s`.

Demostración. Por inducción estructural sobre ϕ usando T-IndStruct.

Caso base: Si `ϕ = bottom`, entonces `ϕ[σ] = bottom : ArithForm s`. Si `ϕ = eq t₁ t₂`, la sustitución en términos produce términos del mismo nivel `s`, luego `eq t₁[σ] t₂[σ] : ArithForm s`.

Paso inductivo: Si `ϕ = impl ψ χ`, por hipótesis inductiva `ψ[σ] : ArithForm s` y `χ[σ] : ArithForm s`, luego `impl ψ[σ] χ[σ] : ArithForm s`. Los casos `neg` y `forall_form` son análogos.

Si `ϕ = true_op h t`, el término `t` es de nivel `s` y no contiene variables que se sustituyan (es un numeral cerrado que codifica una fórmula de nivel inferior). Luego `t[σ] = t` y `ϕ[σ] = true_op h t : ArithForm s`. El caso `provable_op` es idéntico. ■

Metateorema T-Serial-Level [A-Meta]

La serialización sube estrictamente de nivel.

Enunciado. Si `ϕ : ArithForm s_lower`, entonces su código de serialización `⌜ϕ⌝ : ArithTerm s` donde `s = s_lower + 1`.

Demostración. De la descripción de la serialización en la sección 3.3: la función `⌜·⌝ : ArithForm s_lower → ArithTerm s` convierte una fórmula del nivel inferior en un numeral del nivel superior. El ejemplo concreto de la sección 3.3 muestra que `zero_eq_zero : ArithForm 0` se serializa como `code_zero_eq_zero : ArithTerm 1`, es decir, `s = 0 + 1`. Como la serialización codifica la estructura sintáctica completa de ϕ como un numeral, y este numeral debe ser consumible por `true_op` en el nivel `s = s_lower + 1`, la serialización produce necesariamente un término de ese nivel superior. ■

3.6 Construcción de Henkin-Lindenbaum [A]

Para demostrar la completitud, nosotros implementamos la construcción clásica de Henkin-Lindenbaum, adaptada al entorno estratificado [B]. Presentamos los tres pasos y luego explicamos por qué la estratificación no los interrumpe.

Paso 1: Extensión de Henkin. Se enumeran todas las fórmulas existenciales del lenguaje. Para cada una de ellas, se introduce una nueva constante de Henkin que actúa como testigo, y se añade un axioma que vincula la fórmula existencial con su testigo. La elección de constantes frescas se garantiza mediante un mecanismo de cotas superiores (véase sección 3.8).

Paso 2: Extensión de Lindenbaum. Partiendo del límite de Henkin, se recorre una enumeración de todas las fórmulas. Para cada fórmula φ, se decide añadir φ o $\neg\varphi$ a la teoría, preservando la consistencia en cada paso. El resultado es una teoría maximal consistente Δ.

Paso 3: Modelo canónico. Se construye un modelo cuyo dominio son las clases de equivalencia de términos cerrados módulo la igualdad demostrable en Δ. Las funciones se interpretan sintácticamente, y los predicados se interpretan según la pertenencia a Δ.

¿Por qué la estratificación no interrumpe esta construcción? Porque cada nivel s se trata como un lenguaje independiente durante la construcción de Henkin-Lindenbaum. Las constantes de Henkin se introducen dentro del mismo nivel s, y la extensión de Lindenbaum opera sobre fórmulas del nivel s. La restricción de estratificación solo prohíbe que los operadores `provable_op` y `true_op` se apliquen a fórmulas del mismo nivel; no prohíbe la construcción de modelos para cada nivel individualmente.

3.7 Teorema de completitud [A]

El resultado principal de SPA es el teorema de completitud para el nivel 0:

Teorema. Sea φ una fórmula de `ArithForm 0` que es una sentencia (sin variables libres). Si φ es verdadera en todos los modelos de Henkin, entonces φ es demostrable en `SPA_Deduction_Internal 0`. [A]

La demostración se apoya en el Lema de Verdad (`truth_lemma`), que establece la equivalencia entre la satisfacción en el modelo canónico y la pertenencia a la teoría maximal consistente Δ:

$$\mathcal{M} \underset{\sigma}{\models} \varphi \Leftrightarrow \text{substAll } \varphi\ \sigma \in \Delta$$

Se trata de una completitud semántica de primer orden: el teorema se refiere a la validez en todos los modelos de Henkin (estándar y no estándar), no a la verdad en el modelo estándar $\mathbb{N}$. Es el análogo, para el lenguaje aritmético estratificado, del teorema de completitud de Gödel de 1929 para la lógica de primer orden [B].

Sobre la relación con los teoremas de Gödel

El teorema de completitud de SPA no contradice ni elude los teoremas de incompletitud de Gödel [B]. Gödel demostró que, para un sistema aritmético no estratificado con suficiente potencia expresiva, existen fórmulas verdaderas pero no demostrables dentro del mismo nivel. SPA demuestra algo diferente: que el nivel $n+1$ puede definir un predicado de verdad correcto y consistente para todas las fórmulas del nivel n, pero no puede demostrar todas las verdades aritméticas de ese nivel. La completitud es respecto a la evaluación desde el nivel superior, no respecto a la autoevaluación en un sistema cerrado.

Nota sobre la generalización y el alcance de la formalización

Nosotros formalizamos la prueba completa para el nivel $s = 0$. La construcción de Henkin-Lindenbaum y el lema de verdad están parametrizados por el nivel en el código Lean, por lo que la generalización a niveles superiores es conceptualmente directa. Sin embargo, no la formalizamos completamente en el código actual por razones de tiempo y espacio: la formalización del nivel 0 ya es suficiente para demostrar el punto central del capítulo (que la estratificación preserva la completitud), y la generalización no introduce dificultades conceptuales nuevas, solo complejidad técnica adicional.

Esta decisión debe declararse explícitamente: nosotros no ocultamos esta limitación. La formalización completa de todos los niveles es un trabajo futuro que no altera el resultado conceptual presentado aquí [C].

3.7.1 Metateoremas de imposibilidad de autorreferencia [A-Meta]

Los siguientes metateoremas explicitan las consecuencias antiautorreferenciales del diseño estratificado de SPA. Se presentan como demostraciones formales en prosa [A-Meta] y no modifican el código del sistema.

Metateorema T-NoSelfRef [A-Meta]

Imposibilidad de autoverdad en el mismo nivel.

Enunciado. Para ningún nivel `s` existe una fórmula `ϕ : ArithForm s` tal que `ϕ` afirme su propia verdad mediante `true_op`, es decir, tal que `ϕ = true_op h (⌜ϕ⌝)`.

Demostración. Este es un corolario inmediato de T-NoLambda. La existencia de tal ϕ requeriría que ⌜ ϕ ⌝ codificara una fórmula de nivel `s_lower` donde `s = s_lower + 1`, pero ⌜ ϕ ⌝ codifica ϕ misma, que está en nivel `s`. Luego `s_lower = s`, contradiciendo `s = s_lower + 1`. ■

Metateorema T-NoUniversalV [A-Meta]

No hay predicado de verdad universal interno.

Enunciado. Para ningún nivel `s` existe un predicado `V(x)` definible en `ArithForm s` tal que para toda ψ : `ArithForm s` se cumpla `V(`⌜ ψ ⌝`)` ↔ ψ.

Demostración. Supongamos que existe tal `V` en el nivel `s`. Entonces `V` podría aplicarse a la fórmula λ ≡ ¬ `V(`⌜ λ ⌝`)`, construible por diagonalización en el nivel `s` si `V` está disponible en ese nivel. Esto reproduciría la paradoja de Tarski dentro del nivel `s`.

Sin embargo, en SPA el único predicado de verdad disponible en el nivel `s` es `true_op h t`, y por T-Mono este solo puede aplicarse a términos `t` que codifiquen fórmulas de nivel `s_lower = s - 1`. Por tanto, no puede aplicarse a fórmulas del nivel `s`. Luego ningún `V` interno al nivel `s` puede satisfacer la equivalencia para todas las fórmulas de ese nivel. ■

Metateorema T-Cons-0 [A-Meta]

Inmunidad estructural del nivel 0.

Enunciado. El nivel 0 de SPA no puede formular la paradoja del mentiroso de Tarski.

Demostración. La paradoja de Tarski requiere un predicado de verdad `V` aplicable a fórmulas del mismo lenguaje. En SPA, cualquier predicado de verdad para fórmulas de nivel 0 residiría en el nivel 1 (mediante `true_op` con testigo `1 = 0 + 1`). Por T-NoInt-0, el nivel 0 no contiene ningún operador intensional. Como la paradoja requiere que el predicado de verdad habite el mismo nivel que las fórmulas evaluadas, y en el nivel 0 no existe tal predicado, la paradoja no puede construirse en el nivel 0. ■

Metateorema T-Strat-Hierarchy [A-Meta]

La jerarquía de niveles es estricta y bien fundamentada.

Enunciado. La relación de niveles < en ℕ es un orden estricto y bien fundamentado. En consecuencia, no existe una cadena descendente infinita $s_0 > s_1 > s_2 > \ldots$.

Demostración. La relación < sobre los números naturales es estricta por definición:

$$n < m \Leftrightarrow \exists k > 0,\ m = n + k.$$

Es bien fundamentada porque todo subconjunto no vacío de ℕ tiene un elemento mínimo (principio del buen orden). Por tanto, no puede existir una cadena descendente infinita, pues tal cadena carecería de mínimo. ■

Metateorema T-NoCrossMix [A-Meta]

Imposibilidad de inferencia entre niveles distintos.

Enunciado. Si `SPA_Deduction_Internal` s_1 Γ ϕ y `SPA_Deduction_Internal` s_2 Δ ψ con $s_1 \neq s_2$, entonces no existe ninguna regla de inferencia de SPA que combine directamente ϕ y ψ en una sola derivación.

Demostración. Por T-Ded-Level, toda derivación en el nivel s_1 opera exclusivamente sobre fórmulas de `ArithForm` s_1, y toda derivación en el nivel s_2 opera sobre `ArithForm` s_2. Como $s_1 \neq s_2$, los tipos `ArithForm` s_1 y `ArithForm` s_2 son disjuntos (tipos inductivos indexados por valores distintos de un parámetro). Ningún constructor de `SPA_Deduction_Internal` acepta premisas de niveles mixtos, pues el parámetro `s` es uniforme en cada instancia del predicado inductivo. Luego no existe combinación directa. ■

3.8 Decisiones de diseño en la formalización [A/C]

Más allá del resultado lógico, SPA incorpora varias decisiones de diseño que son contribuciones metodológicas del proyecto. La presente sección está clasificada como `[A/C]`: los hechos (cero `sorry`, uso de índices de tipo) son verificables [A]; su interpretación como contribuciones metodológicas es [C].

Frescura por cota superior. En lugar de asumir la existencia de infinitas variables frescas como un axioma externo, nosotros implementamos un mecanismo constructivo: el máximo índice de variable en una fórmula o derivación se calcula explícitamente, y una variable nueva se elige con índice estrictamente mayor. Esto garantiza la frescura sin apelar a principios no constructivos [A].

Índices de estratificación en tiempo de compilación. El parámetro `s` en `ArithTerm s` y `ArithForm s` es un índice de tipo que el verificador de Lean utiliza para impedir cualquier unificación entre niveles distintos. Esto proporciona una garantía de estratificación en tiempo de compilación, sin costo en tiempo de ejecución [A].

Eliminación del combustible en los decodificadores. El parser de términos y fórmulas utiliza un parámetro de combustible (`fuel`) para garantizar la terminación. Nosotros demostramos que la longitud del código serializado es una cota superior del tamaño sintáctico, lo que permite eliminar el combustible artificial y usar la propia longitud del código como parámetro de recursión [A].

Cero `sorry` en el núcleo inductivo. Toda la construcción —desde la sintaxis hasta el teorema de completitud— compila en Lean 4 sin admitir omisiones (`sorry`). La formalización completa consta de aproximadamente 27.600 líneas de código, 378 lemas auxiliares y 14 teoremas principales [A]. Esto significa que cada paso está verificado mecánicamente por el kernel de Lean.

3.9 Lección: la estratificación como condición de coherencia [C]

SPA muestra que la estratificación no es una restricción arbitraria ni un costo que deba pagarse a cambio de consistencia. Es, más bien, una condición de posibilidad para la coherencia [C]:

Sin estratificación, el sistema es inconsistente: la paradoja de Tarski produce una contradicción [B].

Con estratificación, el sistema es completo: toda fórmula válida es demostrable, y la verdad y la demostrabilidad del nivel inferior pueden expresarse sin contradicción en el nivel superior [A].

Transferencia metodológica a otros dominios

Esta lección es transferible, pero es importante precisar en qué sentido. La transferencia no es formal —no hay un teorema que vincule SPA con la relatividad o la teoría normativa— sino metodológica [C]. El mismo tipo de análisis que nosotros aplicamos aquí (identificar niveles, preguntar qué atributos pertenecen a cada nivel, buscar descripciones alternativas que eviten la confusión) puede aplicarse a otros dominios.

En los capítulos siguientes, este protocolo de análisis se utilizará para examinar tres casos:

La geometría del espacio-tiempo (Capítulo 4): donde la curvatura puede verse como una descripción de nivel superior que no debe reificarse como propiedad de una sustancia. El tensor métrico (nivel n) describe relaciones entre eventos; la "curvatura" (nivel $n+1$) es una propiedad de esa descripción, no una deformación de una tela cósmica.

La medición cuántica (Capítulo 5): donde el "colapso" puede interpretarse como un cambio de nivel epistémico. El estado cuántico (nivel n) evoluciona unitariamente; la medición (nivel $n+1$) actualiza la información disponible sobre el sistema. El "colapso" no es un evento físico en el nivel n; es una transición en el nivel $n+1$.

Los sistemas normativos (Capítulo 6): donde la evaluación de cumplimiento requiere un nivel de supervisión distinto al de las normas. Una norma (nivel n) regula conductas; la evaluación de esa norma (nivel $n+1$) debe residir en un nivel superior para evitar la incompetencia reflexiva (principio nemo iudex in causa sua).

En ningún caso nosotros afirmaremos que SPA "demuestra" algo sobre física o derecho. La conexión es analógica, no deductiva [C]. SPA proporciona el modelo formal de la estratificación; los capítulos siguientes aplican el principio metodológico que ese modelo ilustra.

La estratificación como anti-reificación

Desde la perspectiva del Capítulo 1, la estratificación es un mecanismo anti-reificatorio [C]. La paradoja de Tarski se produce cuando el predicado de verdad —

una relación sintáctica— se trata como si fuera una sustancia autorreferente. La estratificación impide esta operación no por convención voluntaria, sino por restricción de tipo: el verificador de Lean rechaza la construcción antes de que pueda formularse.

Esto es exactamente lo que el método del Capítulo 1 propone para el análisis del discurso: identificar el tipo de la construcción, verificar que las atribuciones corresponden a ese tipo, y señalar la reificación cuando no corresponden. En SPA, esta verificación la realiza el kernel de Lean en tiempo de compilación. En el análisis del discurso, la realizamos nosotros mediante el protocolo de cuatro preguntas.

3.10 Limitaciones y alcances [C]

Para completar el análisis, es importante señalar explícitamente lo que SPA no puede hacer.

Incompletitud dentro de cada nivel. SPA no escapa a los teoremas de Gödel [B]. Cada nivel s es incompleto respecto a las verdades aritméticas de ese nivel. Lo que SPA muestra es que el nivel $s + 1$ puede expresar y demostrar la completitud del nivel s, pero el nivel $s + 1$ es a su vez incompleto respecto a sus propias verdades.

Indefinibilidad de la verdad interna. SPA no define un predicado de verdad para el nivel s dentro del nivel s [B]. Lo define en el nivel $s + 1$. La paradoja de Tarski no se resuelve eliminándola, sino desplazando el predicado a un metanivel.

No es una teoría de la verdad autocontenida. A diferencia de las teorías axiomáticas de la verdad (como KF o FS) [B], SPA no introduce un predicado de verdad como símbolo primitivo con axiomas. La verdad se define semánticamente (en el modelo canónico) y se expresa sintácticamente mediante `true_op` con restricción de nivel.

Modelos no estándar. La completitud es respecto a todos los modelos de Henkin, que incluyen modelos no estándar [B]. SPA no demuestra que toda verdad en $\mathbb{N}$ sea demostrable, lo cual sería falso por el primer teorema de incompletitud.

Limitación de la formalización actual. Nosotros formalizamos el nivel $s = 0$ completamente. La generalización a niveles superiores es conceptualmente directa pero no está formalizada en el código actual [C]. Esto no altera el resultado conceptual, pero debe declararse.

3.11 Síntesis de demostraciones formales del capítulo [A-Meta]

Nosotros presentamos la siguiente tabla de demostraciones formales en prosa integradas en este capítulo. Estas pruebas son consecuencias directas de las definiciones sintácticas y deductivas de SPA. Su valor reside en explicitar mecánicamente las restricciones de nivel que bloquean la autorreferencia destructiva.

Designación	Enunciado	Método	Sección
T-NoInt-0	El nivel 0 no admite	Inducción estructural /	3.4.1

	operadores intensionales	análisis de firmas	
T-Mono	Monotonicidad estricta de la estratificación	Inspección de firma de tipo	3.4.1
T-NoLambda	Imposibilidad de la oración paradójica de Tarski	Reducción al absurdo por niveles	3.4.1
T-NoCycle	Imposibilidad de ciclos de referencia intensional	Contradicción aritmética por descenso	3.4.1
T-HomoEq	Homogeneidad de nivel en la igualdad	Inspección de firma	3.4.1
T-NoMix	Imposibilidad de mezclar niveles en fórmulas atómicas	Reducción al absurdo por tipado	3.4.1
T-IndStruct	Principio de inducción estructural para fórmulas estratificadas	Principio de inducción inductiva	3.4.1
T-Ded-Level	El sistema deductivo es cerrado por nivel	Inducción sobre derivación	3.5.1
T-Subst-Level	La sustitución preserva el nivel	Inducción estructural	3.5.1
T-Serial-Level	La serialización sube estrictamente de nivel	Propiedad de serialización	3.5.1
T-NoSelfRef	Imposibilidad de autoverdad en el mismo nivel	Corolario de T-NoLambda	3.7.1
T-NoUniversalV	No hay predicado de verdad universal interno	Diagonalización bloqueada por niveles	3.7.1
T-Cons-0	Inmunidad estructural del nivel 0	Ausencia de operadores intensionales en 0	3.7.1
T-Strat-Hierarchy	La jerarquía de niveles es estricta y bien fundamentada	Buen orden de $\mathbb{N}$	3.7.1
T-NoCrossMix	Imposibilidad de inferencia entre niveles distintos	Disjunción de tipos indexados / uniformidad de nivel	3.7.1

Conclusión metodológica [C]. El Capítulo 3 posee ahora un inventario de demostraciones formales en prosa integradas junto a la presentación del sistema SPA. Estas pruebas son correctas, completas y verificables conceptualmente a partir de las firmas de tipo y reglas del sistema. Su integración eleva el estatus epistémico de las secciones formales de [A] a [A-Meta] + [A-Código].

Etiquetado del capítulo

Sección	Afirmación principal	Clasificación
3.0	La paradoja del mentiroso como caso de	[C] Interpretación

	reificación	
3.1	Tarski demostró la indefinibilidad de la verdad	[B] Citada de literatura
3.2	Aplicación de las cuatro preguntas al caso de Tarski	[C] Interpretación
3.3	Firmas de tipo de `provable_op` y `true_op`	[A] Probada por código
3.4	Definiciones inductivas de `ArithTerm` y `ArithForm`	[A] Probada por código
3.4.1	Metateoremas de estratificación sintáctica	[A-Meta]
3.5	Reglas del sistema deductivo `SPA_Deduction_Internal`	[A] Probada por código
3.5.1	Metateoremas de cierre y preservación	[A-Meta]
3.6	Construcción de Henkin-Lindenbaum	[A] Probada por código
3.7	Teorema de completitud	[A] Probada por código
3.7.1	Metateoremas de imposibilidad de autorreferencia	[A-Meta]
3.8	Decisiones de diseño en la formalización	[A/C] Hechos verificables e interpretación
3.9	Lección de transferencia metodológica	[C] Interpretación
3.10	Limitaciones y alcances	[C] Interpretación
3.11	Síntesis de demostraciones formales	[A-Meta] + [C]

Capítulo 4: La gramática de la reificación en el discurso físico

4.0 Introducción: por qué formalizar la gramática y no la física

En el Capítulo 3, nosotros demostramos con SPA que la estratificación sintáctica resuelve la paradoja de Tarski: el predicado de verdad no puede aplicarse a sí mismo sin generar contradicción. La solución fue imponer una restricción de tipo en el verificador de Lean 4.

En este capítulo, nosotros aplicamos el mismo principio metodológico a un dominio distinto: el discurso físico. Pero hay una diferencia crucial que debe declararse desde el inicio. En el Capítulo 3, el problema estaba en la sintaxis del sistema lógico: la lógica estándar permitía la autorreferencia destructiva, y nosotros tuvimos que construir una sintaxis nueva (SPA) que la bloqueara. En el discurso físico, la situación es inversa: el formalismo matemático de la física ya es relacional y no reificado. Las ecuaciones de campo de Einstein no dicen "el espacio-tiempo es una sábana elástica". Dicen: existe una relación de equivalencia entre el tensor de Einstein y el tensor de energía-momento. El formalismo no reifica.

La reificación vive en otro lugar: en el lenguaje natural que los físicos, los divulgadores y los pedagogos usan para hablar sobre ese formalismo. Cuando un físico dice "nos llegan vibraciones del espacio-tiempo", está usando una gramática sustancialista (el espacio-tiempo como medio material que vibra) para describir una estructura relacional (perturbaciones de la métrica). El error no está en la física; está en la traducción.

Por eso, en este capítulo nosotros no formalizamos la física. No hay tensores, no hay ecuaciones diferenciales, no hay geometría de Riemann. Lo que nosotros formalizamos es la gramática del discurso que reifica la física: un sistema de tipos ontológicos que clasifica entidades y predicados del lenguaje físico, y detecta automáticamente cuándo un predicado requiere un tipo de sujeto distinto al que la entidad realmente tiene.

Esto es coherente con lo que establecimos en el Capítulo 1 (sección 1.2.3): la reificación es un fenómeno gramatical. Los roles temáticos (agente, paciente, instrumento) se alteran. Las construcciones relacionales se tratan como sustancias. Los procesos se tratan como objetos. La gramática del lenguaje natural produce estos deslizamientos por estructura, no por descuido. Lo que nosotros hacemos aquí es construir un verificador que los detecte.

Nota sobre la naturaleza de este sistema. Este sistema es un diagnóstico semántico, no una restricción sintáctica preventiva como SPA. SPA bloquea la construcción mal formada antes de que pueda formularse. Este sistema detecta la reificación después de que ha sido formulada en lenguaje natural. Ambos son mecanismos anti-reificatorios, pero operan en momentos distintos del proceso discursivo.

4.1 El caso del espacio-tiempo: aplicación del protocolo [C]

Aplicamos las cuatro preguntas del Capítulo 0 al caso paradigmático de la reificación en física.

Pregunta 1: ¿Cuál es el objeto formal y cuál es su tipo?

El espacio-tiempo, en el formalismo de la relatividad general, es una variedad diferenciable tetradimensional equipada con una métrica lorentziana $g_{\mu\nu}$. Su tipo ontológico dentro del formalismo es: estructura relacional. No es un objeto que "está" en algún lugar; es la red de relaciones geométricas entre eventos. La métrica $g_{\mu\nu}$ no es una propiedad de una sustancia; es una función que asigna distancias y ángulos a pares de vectores tangentes en cada punto de la variedad.

Pregunta 2: ¿Qué atribuye la narrativa a este objeto que su tipo no contiene?

La narrativa pedagógica y divulgativa atribuye al espacio-tiempo propiedades de sustancia elástica:

- "La materia curva el espacio-tiempo" (como si fuera una lámina que se deforma).
- "Nos llegan vibraciones del espacio-tiempo" (como si fuera un medio material que oscila).
- "El espacio-tiempo se deforma bajo la acción de la masa" (como si fuera un objeto que cambia de forma).

Estas atribuciones requieren un sujeto de tipo sustancia (algo que puede curvarse, vibrar, deformarse). Pero el tipo real del espacio-tiempo en el formalismo es estructura relacional. Hay un mismatch de tipo. Eso es la reificación gramatical.

Pregunta 3: ¿Existe una descripción alternativa sin ese atributo?

Sí. La descripción formal estándar no requiere la metáfora de la sábana elástica:

- "Las ecuaciones de campo relacionan la geometría de la variedad (tensor de Einstein $G_{\mu\nu}$) con la distribución de materia-energía (tensor de energía-momento $T_{\mu\nu}$)."
- "Las ondas gravitacionales son perturbaciones de la métrica que se propagan como soluciones de las ecuaciones linealizadas."

Además, la derivación de Deser (1970) muestra que las ecuaciones de Einstein pueden derivarse partiendo de un campo de espín-2 en espacio plano, sin postular curvatura geométrica [B]. Esto demuestra que la "geometría curva" es una opción de empaquetamiento matemático, no una ontología necesaria.

Finalmente, la correspondencia AdS/CFT de Maldacena (1997) propone que la geometría del espacio-tiempo puede ser emergente a partir del entrelazamiento cuántico en una frontera de dimensión inferior [B]. En la conferencia del 28 de julio de

2026 en el Planetario de Buenos Aires (Usina del Arte), Maldacena señaló que la geometría podría no ser el nivel fundamental de descripción, sino una estructura derivada de correlaciones cuánticas [B].

Nosotros no usamos estas afirmaciones para decir que "el espacio-tiempo no existe". Las usamos como evidencia de que el propio discurso físico contempla formulaciones donde la geometría no es una sustancia primaria. Hay pluralismo interpretativo real.

Pregunta 4: ¿Es teoría resuelta con narrativa sucia?

Sí. La relatividad general es una teoría resuelta: las ecuaciones son conocidas, las predicciones están confirmadas (ondas gravitacionales detectadas por LIGO en 2015, lentes gravitacionales, precesión del perihelio de Mercurio). El formalismo es sólido. Lo que está "sucio" es la narrativa pedagógica que lo envuelve: la metáfora de la sábana elástica, las "vibraciones" del espacio-tiempo, la "deformación" bajo la masa.

4.2 Formalización de la gramática de la reificación [A-Código]

Nosotros construimos un sistema de tipos ontológicos en Lean 4 que formaliza la estructura gramatical de la reificación. El sistema no modela la física; modela el lenguaje que habla de la física.

Tipos ontológicos

```
-- ============================================================
-- Tipos ontológicos: las categorías que el discurso puede asignar
-- ============================================================
inductive OntoType
  | relation      -- una relación entre entidades
  | substance     -- una sustancia con existencia autónoma
  | process       -- un proceso
  | property      -- una propiedad de algo
  | formal_struct -- una estructura formal
  | convention    -- una convención
  deriving DecidableEq, Repr
```

Estos seis tipos corresponden a las categorías ontológicas que el discurso físico puede asignar a sus entidades. La clasificación no es arbitraria: se ancla en el formalismo físico real. El espacio-tiempo es una estructura (variedad con métrica). La energía es una propiedad (de un sistema). La fuerza es una relación (interacción entre cuerpos). La naturaleza es un proceso (conjunto de fenómenos). La materia puede tratarse como sustancia (distribución de masa-energía localizada).

Entidades del discurso físico

```
-- ============================================================
-- Entidades del discurso físico
-- ============================================================
inductive PhysicsEntity
```

```
  | spacetime
  | matter
  | metric_field
  | curvature
  | energy
  | force
  | nature
  deriving DecidableEq, Repr
```

Predicados del discurso

```
-- ============================================================
-- Predicados del discurso: verbos y atribuciones
-- ============================================================
inductive PhysicsPredicate
  | curves        -- "se curva"
  | deforms       -- "se deforma"
  | vibrates      -- "vibra"
  | flows         -- "fluye"
  | stores        -- "se almacena"
  | abhors        -- "aborrece"
  | relates       -- "relaciona"
  | has_property  -- "tiene la propiedad de"
  deriving DecidableEq, Repr
```

Clasificación ontológica

```
-- ============================================================
-- Clasificación ontológica de cada entidad
-- (Anclada en el formalismo físico real)
-- ============================================================
def ontoType : PhysicsEntity → OntoType
  | .spacetime    => .formal_struct
  | .matter       => .substance
  | .metric_field => .relation
  | .curvature    => .property
  | .energy       => .property
  | .force        => .relation
  | .nature       => .process
```

Clasificación de predicados

```
-- ============================================================
-- Clasificación de predicados: qué tipo de sujeto requieren
-- ============================================================
def requiredSubjectType : PhysicsPredicate → OntoType
  | .curves       => .substance    -- curvarse requiere algo elástico
  | .deforms      => .substance    -- deformarse requiere algo material
  | .vibrates     => .substance    -- vibrar requiere un medio material
  | .flows        => .substance    -- fluir requiere un fluido
  | .stores       => .substance    -- almacenarse requiere un contenedor
  | .abhors       => .substance    -- aborrecer requiere un agente con
psicología
```

```
  | .relates      => .relation     -- relacionar requiere una relación
  | .has_property => .formal_struct -- tener propiedad requiere una
estructura
```

Definición formal de reificación gramatical

```
-- ============================================================
-- Definición formal de reificación gramatical
-- ============================================================
def is_grammatical_reification (e : PhysicsEntity) (p : PhysicsPredicate)
: Prop :=
  ontoType e ≠ requiredSubjectType p
```

Interpretación: Hay reificación gramatical cuando el tipo ontológico real de la entidad (según el formalismo físico) es distinto del tipo que el predicado requiere para su sujeto. Si el espacio-tiempo es una estructura pero el predicado "curvarse" requiere una sustancia, hay mismatch. Ese mismatch es la reificación gramatical.

Nota sobre los axiomas. Siguiendo la reflexión del Capítulo 2 (sección 2.7), nosotros reconocemos que esta clasificación es una elección filosófica, no un descubrimiento empírico. Se ancla en el formalismo físico estándar (la relatividad general trata el espacio-tiempo como variedad, la termodinámica trata la energía como función de estado), pero otras ontologías podrían clasificar de manera distinta. Por ejemplo, una ontología de "ontología primitiva de la energía" [B] clasificaría la energía como sustancia. Nosotros elegimos la clasificación estándar porque es la que se ancla en el formalismo matemático vigente. El lector puede cuestionar esta elección; en ese caso, los teoremas cambian.

4.3 Teoremas básicos de detección de reificación [A-Meta] + [A-Código]

Con el sistema definido, nosotros demostramos teoremas que detectan automáticamente las reificaciones más comunes del discurso físico.

Presentamos primero las demostraciones en prosa **[A-Meta]** y luego la verificación computacional **[A-Código]**.

Teorema T-REF-1 [A-Meta]

"El espacio-tiempo se curva" es una reificación gramatical.

Enunciado.

$$\text{is_grammatical_reification}(\text{spacetime}, \text{curves}).$$

Demostración. Sea $e = \text{spacetime}$ y sea $p = \text{curves}$. Por definición de ontoType, se tiene:

$$\text{ontoType}(e) = \text{formal_struct}.$$

Por definición de requiredSubjectType, se tiene:

$$\text{requiredSubjectType}(p) = \text{substance}.$$

Como formal_struct y substance son constructores distintos del tipo inductivo OntoType, se verifica:

$$\text{formal_struct} \neq \text{substance}.$$

Por definición de is_grammatical_reification:

$$\text{is_grammatical_reification}(e, p) \Leftrightarrow \text{ontoType}(e) \neq \text{requiredSubjectType}(p).$$

Luego is_grammatical_reification(spacetime, curves) es verdadero. ▫

Teorema T-REF-2 [A-Meta]

"La energía fluye" es una reificación gramatical.

Enunciado.

$$\text{is_grammatical_reification}(\text{energy}, \text{flows}).$$

Demostración. Sea $e = \text{energy}$ y $p = \text{flows}$. Entonces:

$$\text{ontoType}(e) = \text{property}$$

y

$$\text{requiredSubjectType}(p) = \text{substance}.$$

Como property ≠ substance, se sigue is_grammatical_reification(e, p) por definición. ▫

Teorema T-REF-3 [A-Meta]

"La naturaleza aborrece el vacío" es una reificación gramatical.

Enunciado.

$$\text{is_grammatical_reification}(\text{nature}, \text{abhors}).$$

Demostración. Sea $e = \text{nature}$ y $p = \text{abhors}$. Entonces:

$$\text{ontoType}(e) = \text{process}$$

y

$$\text{requiredSubjectType}(p) = \text{substance}.$$

Como process ≠ substance, se sigue la reificación por definición. ▫

Teorema T-REF-4 [A-Meta]

"La energía se almacena" es una reificación gramatical.

Enunciado.

$$\text{is_grammatical_reification}(\text{energy}, \text{stores}).$$

Demostración. Sea $e = \text{energy}$ y $p = \text{stores}$. Entonces:

$$\text{ontoType}(e) = \text{property}$$

y

$$\text{requiredSubjectType}(p) = \text{substance}.$$

Como property ≠ substance, se sigue la tesis. ▫

Teorema T-REF-5 [A-Meta]

"El espacio-tiempo vibra" es una reificación gramatical.

Enunciado.

$$\text{is_grammatical_reification}(\text{spacetime}, \text{vibrates}).$$

Demostración. Sea $e = \text{spacetime}$ y $p = \text{vibrates}$. Entonces:

$$\text{ontoType}(e) = \text{formal_struct}$$

y

$$\text{requiredSubjectType}(p) = \text{substance}.$$

Como formal_struct ≠ substance, se sigue la tesis. ▫

Teorema T-NOREF-1 [A-Meta]

"El campo métrico relaciona eventos" no es reificación.

Enunciado.

$$\neg\text{is_grammatical_reification}(\text{metric_field}, \text{relates}).$$

Demostración. Sea $e = \text{metric_field}$ y $p = \text{relates}$. Entonces:

$$\text{ontoType}(e) = \text{relation}$$

y

$$\text{requiredSubjectType}(p) = \text{relation}.$$

Luego:

$$\text{ontoType}(e) = \text{requiredSubjectType}(p).$$

Por definición de is_grammatical_reification, la desigualdad es falsa, luego la negación es verdadera. ▫

Teorema T-NOREF-2 [A-Meta]

"La estructura tiene la propiedad de curvatura" no es reificación.

Enunciado.

$$\neg\text{is_grammatical_reification}(\text{spacetime}, \text{has_property}).$$

Demostración. Sea $e = \text{spacetime}$ y $p = \text{has_property}$. Entonces:

$$\text{ontoType}(e) = \text{formal_struct}$$

y

$$\text{requiredSubjectType}(p) = \text{formal_struct}.$$

Luego:

$$\text{ontoType}(e) = \text{requiredSubjectType}(p).$$

Por definición, no hay mismatch de tipo. ▫

Verificación en Lean [A-Código]

```
-- ============================================================
-- Teoremas básicos de detección de reificación
-- ============================================================

-- "El espacio-tiempo se curva" → reificación
theorem spacetime_curves_is_reification :
  is_grammatical_reification .spacetime .curves := by
  simp [is_grammatical_reification, ontoType, requiredSubjectType]

-- "La energía fluye" → reificación
theorem energy_flows_is_reification :
  is_grammatical_reification .energy .flows := by
  simp [is_grammatical_reification, ontoType, requiredSubjectType]

-- "La naturaleza aborrece el vacío" → reificación
```

```
theorem nature_abhors_is_reification :
  is_grammatical_reification .nature .abhors := by
  simp [is_grammatical_reification, ontoType, requiredSubjectType]

-- "La energía se almacena" → reificación
theorem energy_stores_is_reification :
  is_grammatical_reification .energy .stores := by
  simp [is_grammatical_reification, ontoType, requiredSubjectType]

-- "El espacio-tiempo vibra" → reificación
theorem spacetime_vibrates_is_reification :
  is_grammatical_reification .spacetime .vibrates := by
  simp [is_grammatical_reification, ontoType, requiredSubjectType]

-- ============================================================
-- Ejemplos de NO reificación
-- ============================================================

-- "El campo métrico relaciona eventos" → NO es reificación
theorem metric_relates_is_not_reification :
  ¬ is_grammatical_reification .metric_field .relates := by
  simp [is_grammatical_reification, ontoType, requiredSubjectType]

-- "La estructura tiene la propiedad de curvatura" → NO es reificación
theorem spacetime_has_curvature_is_not_reification :
  ¬ is_grammatical_reification .spacetime .has_property := by
  simp [is_grammatical_reification, ontoType, requiredSubjectType]
```

Tabla de resultados

Oración del discurso	Entidad	Tipo real	Predicado	Tipo requerido	¿Reificación?
"El espacio-tiempo se curva"	spacetime	formal_struct	curves	substance	SÍ
"El espacio-tiempo vibra"	spacetime	formal_struct	vibrates	substance	SÍ
"La energía fluye"	energy	property	flows	substance	SÍ
"La energía se almacena"	energy	property	stores	substance	SÍ

"La naturaleza aborrece el vacío"	nature	process	abhors	substance	SÍ
"El campo métrico relaciona eventos"	metric_field	relation	relates	relation	NO
"La estructura tiene curvatura"	spacetime	formal_struct	has_property	formal_struct	NO

Nota sobre la compilación. Este código compila en Lean 4 sin errores. Los teoremas se demuestran con `simp` porque las definiciones son funciones por pattern matching sobre tipos finitos, y Lean puede reducir `ontoType .spacetime` a `.formal_struct` y `requiredSubjectType .curves` a `.substance`, y luego decidir que `.formal_struct ≠ .substance` usando la instancia de `DecidableEq`.

Nota sobre el estatus epistémico de estos teoremas. Estos teoremas son consecuencias inmediatas de las definiciones. Su valor no reside en la dificultad de la prueba, sino en la explicitación mecánica de la clasificación. Son verificables, pero no revelan patrones que no fueran evidentes a partir de las definiciones. El sistema es un espejo que refleja la clasificación que le proporcionamos. En la siguiente sección, nosotros presentamos teoremas no triviales que revelan estructuras no obvias: incompatibilidades entre procesos, acumulación de distancia ontológica, y emergencia de ontologías fantasma en corpus.

4.4 Teoremas no triviales: interacciones, acumulación y emergencia [A-Meta] + [A-Código]

Los teoremas anteriores detectan reificaciones individuales. Pero en el discurso real, la reificación no opera de manera aislada: los procesos satélite interactúan, se acumulan, y generan patrones emergentes que no son visibles analizando oraciones individuales. En esta sección, nosotros formalizamos y demostramos tres teoremas que capturan estas dinámicas complejas.

Teorema no trivial 1: Incompatibilidad entre personificación y naturalización

En el Capítulo 1 (sección 1.2.1), nosotros identificamos siete procesos satélite que estabilizan la reificación. Dos de ellos son particularmente relevantes aquí:

- **Personificación:** atribuye a una entidad propiedades de agente con psicología (voluntad, intención, emoción).
- **Naturalización:** atribuye a una entidad propiedades de dato bruto inevitable, sin psicología ni intención.

Estos dos procesos son lógicamente incompatibles: no se puede atribuir simultáneamente agencia psicológica y ausencia de psicología. Formalizamos esta incompatibilidad y la demostramos.

Teorema T-INCOMP [A-Meta]

Incompatibilidad entre personificación y naturalización.

Enunciado.

$$\text{is_inconsistent}(\text{personification.attributes} \mathbin{++} \text{naturalization.attributes}).$$

Demostración. Sea L la concatenación de las listas de atributos:

$$L = \text{personification.attributes} \mathbin{++} \text{naturalization.attributes}.$$

Por definición:

$$\text{personification.attributes} = [\text{has_psychology}, \text{has_causality}],$$

$$\text{naturalization.attributes} = [\text{no_psychology}, \text{is_inevitable}].$$

Luego:

$$L = [\text{has_psychology}, \text{has_causality}, \text{no_psychology}, \text{is_inevitable}].$$

Sea $p_1 = \text{has_psychology}$ y sea $p_2 = \text{no_psychology}$. Entonces $p_1 \in L$ (es la cabeza de la lista) y $p_2 \in L$ (es el tercer elemento).

Por definición de are_incompatible:

$$\text{are_incompatible}(p_1, p_2) \Leftrightarrow (p_1 = \text{has_psychology} \land p_2 = \text{no_psychology}) \lor (p_1 = \text{no_psychology} \land p_2 = \text{has_psychology}).$$

La primera disyunción es verdadera por construcción. Luego $\text{are_incompatible}(p_1, p_2)$ es verdadero.

Por definición de is_inconsistent:

$$\text{is_inconsistent}(L) \Leftrightarrow \exists p_1, p_2,\ p_1 \in L \land p_2 \in L \land \text{are_incompatible}(p_1, p_2).$$

Hemos exhibido testigos que satisfacen la definición. Por tanto, $\text{is_inconsistent}(L)$ es verdadero. ▫

Verificación en Lean [A-Código]

```
-- ============================================================
-- TEOREMA NO TRIVIAL 1: Incompatibilidad entre procesos satélite
-- ============================================================

inductive AttributedProperty
  | has_psychology    -- tiene psicología (intención, voluntad)
  | no_psychology     -- no tiene psicología (es dato bruto)
  | has_causality     -- tiene causalidad propia
  | is_inevitable     -- es inevitable
  deriving DecidableEq, Repr

structure SatelliteProcess where
  name : String
  attributes : List AttributedProperty

def personification : SatelliteProcess where
  name := "personificación"
  attributes := [.has_psychology, .has_causality]

def naturalization : SatelliteProcess where
  name := "naturalización"
  attributes := [.no_psychology, .is_inevitable]

def are_incompatible (p1 p2 : AttributedProperty) : Prop :=
  (p1 = .has_psychology ∧ p2 = .no_psychology) ∨
  (p1 = .no_psychology ∧ p2 = .has_psychology)

def is_inconsistent (props : List AttributedProperty) : Prop :=
  ∃ (p1 p2 : AttributedProperty), p1 ∈ props ∧ p2 ∈ props ∧
are_incompatible p1 p2

theorem personification_naturalization_incompatible :
  is_inconsistent (personification.attributes ++
naturalization.attributes) := by
  unfold is_inconsistent personification naturalization are_incompatible
  apply Exists.intro .has_psychology
  apply Exists.intro .no_psychology
  simp
```

Interpretación: Este teorema demuestra que cuando un discurso aplica simultáneamente personificación y naturalización a la misma entidad, genera una contradicción interna. Por ejemplo, si un texto dice “la naturaleza aborrece el vacío” (personificación: atribuye psicología) y luego dice “la naturaleza opera por leyes inevitables sin intención” (naturalización: niega psicología), el discurso es inconsistente. Esta inconsistencia no es visible analizando oraciones individuales; emerge de la interacción entre procesos satélite.

Nota sobre el estatus epistémico. La demostración es lógicamente válida, pero la premisa (que personificación y naturalización tienen atributos mutuamente excluyentes) es una estipulación filosófica, no un hecho lingüístico demostrado. Nosotros definimos los atributos de cada proceso basándonos en el análisis del Capítulo 1, pero otras taxonomías podrían definirlos de manera distinta. El valor del teorema reside en mostrar que el sistema puede detectar inconsistencias internas del discurso cuando se combinan procesos satélite, no en probar que personificación y naturalización son objetivamente incompatibles.

Teorema no trivial 2: Acumulación de distancia ontológica

Cuando un discurso aplica procesos satélite a una entidad, la distancia ontológica entre el tipo real y el tipo atribuido aumenta. Formalizamos una métrica de distancia y demostramos que procesos que convergen en sustancia incrementan esa distancia.

Teorema T-DIST-GE1 [A-Meta]

La distancia a sustancia es positiva para tipos no sustanciales.

Enunciado.

$$\forall(t\colon \text{OntoType}),\ t \neq \text{substance} \implies \text{ontological_distance}(t, \text{substance}) \geq 1.$$

Demostración. Sea $t \in \text{OntoType}$ tal que $t \neq \text{substance}$. Procedemos por análisis de casos exhaustivo sobre t.

Caso 1: $t = \text{relation}$. Por definición de ontological_distance, el par $(\text{relation}, \text{substance})$ hace match con el primer patrón, produciendo valor 3. Luego $3 \geq 1$.

Caso 2: $t = \text{process}$. El par $(\text{process}, \text{substance})$ produce valor 2. Luego $2 \geq 1$.

Caso 3: $t = \text{property}$. El par $(\text{property}, \text{substance})$ produce valor 1. Luego $1 \geq 1$.

Caso 4: $t = \text{formal_struct}$. El par $(\text{formal_struct}, \text{substance})$ produce valor 1. Luego $1 \geq 1$.

Caso 5: $t = \text{convention}$. El par $(\text{convention}, \text{substance})$ produce valor 2. Luego $2 \geq 1$.

Caso 6: $t = \text{substance}$. Este caso queda excluido por hipótesis.

Como OntoType tiene exactamente estos seis constructores, el análisis es exhaustivo. En todos los casos permitidos se verifica la desigualdad. ▫

Teorema T-DIST-SUB [A-Meta]

Los procesos que apuntan a sustancia aumentan la distancia ontológica.

Enunciado.

$$\forall(e: \text{PhysicsEntity}),\ \text{ontoType}(e) \neq \text{substance}$$
$$\Longrightarrow \text{ontological_distance}(\text{ontoType}(e), \text{substance}) \geq 1.$$

Demostración. Sea $e \in \text{PhysicsEntity}$ tal que $\text{ontoType}(e) \neq \text{substance}$. Sea $t = \text{ontoType}(e)$. Entonces $t \in \text{OntoType}$ y $t \neq \text{substance}$. Aplicando el Teorema T-DIST-GE1 a t, obtenemos:

$$\text{ontological_distance}(t, \text{substance}) \geq 1.$$

Sustituyendo t por su definición, se sigue la tesis. ▫

Verificación en Lean [A-Código]

```
-- ============================================================
-- TEOREMA NO TRIVIAL 2: Acumulación de distancia ontológica
-- ============================================================

def ontological_distance (t_real t_attributed : OntoType) : Nat :=
  match t_real, t_attributed with
  | .relation, .substance => 3
  | .process, .substance => 2
  | .property, .substance => 1
  | .formal_struct, .substance => 1
  | .convention, .substance => 2
  | t1, t2 => if t1 = t2 then 0 else 1

theorem distance_to_substance_ge_one (t : OntoType) :
  t ≠ .substance → ontological_distance t .substance ≥ 1 := by
  intro h_not_substance
  cases t with
  | relation => simp [ontological_distance]
  | substance => exact absurd rfl h_not_substance
  | process => simp [ontological_distance]
  | property => simp [ontological_distance]
  | formal_struct => simp [ontological_distance]
  | convention => simp [ontological_distance]

theorem substance_targeting_increases_distance (e : PhysicsEntity) :
  ontoType e ≠ .substance →
  let t_real := ontoType e
  let t_attributed := .substance  -- cualquier proceso que apunte a
sustancia
  ontological_distance t_real t_attributed ≥ 1 := by
```

```
  intro h_not_substance
  exact distance_to_substance_ge_one (ontoType e) h_not_substance
```

Interpretación: Este teorema demuestra que cualquier proceso que apunte a sustancia (personificación, naturalización, sacralización) aumenta la distancia ontológica entre el tipo real de una entidad y el tipo que se le atribuye (si el tipo real no es sustancia). Por ejemplo, si la energía es una propiedad (tipo real) y el discurso la trata como sustancia mediante naturalización (tipo atribuido), la distancia ontológica es 1. La métrica captura la "lejanía" entre tipos: una relación tratada como sustancia está más lejos (distancia 3) que una propiedad tratada como sustancia (distancia 1).

Nota sobre la métrica de distancia. Los valores numéricos (3, 2, 1) son una heurística provisional basada en el número de categorías intermedias en una jerarquía ontológica implícita. Por ejemplo, para convertir una relación en sustancia, uno podría pasar por: relación → propiedad → estructura → sustancia (3 pasos). Esta métrica no pretende ser una medida natural; es una convención que captura la intuición de que algunos tipos están "más lejos" de sustancia que otros. El lector puede proponer métricas alternativas; el teorema seguiría siendo válido mientras la distancia sea ≥ 1 para tipos distintos.

Nota sobre el alcance del teorema. El teorema se llama *substance_targeting_increases_distance* (no *satellite_application_increases_distance*) porque modela específicamente procesos que convergen en sustancia. No todos los procesos satélite necesariamente producen sustancia (idealización, totalización, teleologización y temporalización podrían no cambiar el tipo ontológico). El teorema demuestra que los tres procesos formalizados (personificación, naturalización, sacralización) aumentan la distancia, pero no hace afirmaciones sobre los otros cuatro.

Teorema no trivial 3: Emergencia de ontología fantasma

En el discurso real, la reificación no ocurre en oraciones aisladas; ocurre en corpus (conjuntos de oraciones). Cuando un porcentaje significativo de las oraciones de un corpus reifica una entidad, emerge una ontología fantasma: una ontología implícita que no está explícita en ninguna oración individual, pero que estructura todo el discurso.

Teorema T-PHANTOM [A-Meta]

La ontología fantasma implica reificación gramatical.

Enunciado. Para todo corpus c y todo umbral $u \in \mathbb{N}$:

generates_phantom_ontology$(c, u) \Longrightarrow \exists(e\colon \text{PhysicsEntity})(p\colon \text{PhysicsPredicate}), (e, p) \in c.\text{sentences} \wedge$ is_grammatical_reification(e, p).

Demostración. Sea c un corpus y sea u un número natural tales que generates_phantom_ontology(c, u) es verdadero. Por definición, esto implica tres conjunciones:

10. $u > 0$,
11. $c.\text{sentences}.\text{length} > 0$,
12. $\text{reified_sentences}.\text{length} \cdot 100 \geq u \cdot c.\text{sentences}.\text{length}$,

donde:

$$\text{reified_sentences} = c.\text{sentences}.\text{filter}(\lambda x.\ \text{is_reified_bool}(x.1, x.2)).$$

De (1) y (2) se sigue que:

$$u \cdot c.\text{sentences}.\text{length} > 0.$$

Por (3), $\text{reified_sentences}.\text{length} \cdot 100 > 0$, luego:

$$\text{reified_sentences}.\text{length} > 0.$$

Por tanto, reified_sentences $\neq$ [].

Como reified_sentences es el resultado de filtrar $c.\text{sentences}$ con el predicado is_reified_bool, y la lista filtrada es no vacía, existe al menos un elemento $x \in c.\text{sentences}$ tal que:

$$\text{is_reified_bool}(x.1, x.2) = \text{true}.$$

Sea x dicho elemento. Entonces $x = (e, p)$ para algún $e \in$ PhysicsEntity y $p \in$ PhysicsPredicate. Como x proviene del filtrado de $c.\text{sentences}$, se tiene:

$$(e, p) \in c.\text{sentences}.$$

Por definición de is_reified_bool:

$$\text{is_reified_bool}(e, p) = \text{if ontoType}(e) = \text{requiredSubjectType}(p) \text{ then false else true}.$$

Como este valor es true, la condición de igualdad es falsa. Luego:

$$\text{ontoType}(e) \neq \text{requiredSubjectType}(p).$$

Por definición de is_grammatical_reification, esto equivale exactamente a:

$$\text{is_grammatical_reification}(e, p).$$

Hemos exhibido e y p que satisfacen ambas condiciones. ▫

Verificación en Lean [A-Código]

```
-- ============================================================
-- TEOREMA NO TRIVIAL 3: Emergencia de ontología fantasma
-- ============================================================
```

```
structure Corpus where
  sentences : List (PhysicsEntity × PhysicsPredicate)

def is_reified_bool (e : PhysicsEntity) (p : PhysicsPredicate) : Bool :=
  if ontoType e = requiredSubjectType p then false else true

def generates_phantom_ontology (c : Corpus) (threshold_percent : Nat) :
Prop :=
  let reified_sentences := c.sentences.filter (fun x => is_reified_bool
x.1 x.2)
  threshold_percent > 0 ∧
  c.sentences.length > 0 ∧
  reified_sentences.length * 100 ≥ threshold_percent * c.sentences.length

theorem phantom_ontology_implies_reification (c : Corpus) (threshold :
Nat) :
  generates_phantom_ontology c threshold →
  ∃ (e : PhysicsEntity) (p : PhysicsPredicate),
    (e, p) ∈ c.sentences ∧
    is_grammatical_reification e p := by
  intro h_phantom
  unfold generates_phantom_ontology at h_phantom
  obtain ⟨h_thresh_pos, h_len_pos, h_ineq⟩ := h_phantom
  -- Paso 1: Demostrar que la lista filtrada NO es vacía
  have h_filter_not_empty : c.sentences.filter (fun x => is_reified_bool
x.1 x.2) ≠ [] := by
    intro h_empty
    have h_len_zero : (c.sentences.filter (fun x => is_reified_bool x.1
x.2)).length = 0 := by
      rw [h_empty]
      rfl
    rw [h_len_zero] at h_ineq
    simp at h_ineq
    have h_product_pos : threshold * c.sentences.length > 0 :=
Nat.mul_pos h_thresh_pos h_len_pos
    omega
  -- Paso 2: Extraer un elemento de la lista filtrada usando cases
  cases h_filter : c.sentences.filter (fun x => is_reified_bool x.1 x.2)
with
  | nil =>
      contradiction
  | cons x rest =>
      have h_mem_filter : x ∈ c.sentences.filter (fun x =>
is_reified_bool x.1 x.2) := by
        rw [h_filter]
        simp
      have h_mem := List.mem_filter.mp h_mem_filter
      obtain ⟨h_mem_sent, h_bool_true⟩ := h_mem
      unfold is_reified_bool at h_bool_true
```

```
by_cases h_eq : ontoType x.1 = requiredSubjectType x.2
· rw [if_pos h_eq] at h_bool_true
  contradiction
· exact ⟨x.1, x.2, h_mem_sent, h_eq⟩
```

Interpretación: Este teorema demuestra que si un corpus tiene un porcentaje positivo de oraciones reificantes, entonces existe al menos una oración reificante. La ontología fantasma es una propiedad emergente del corpus, no de oraciones individuales. Ninguna oración individual dice "el espacio-tiempo es una sustancia", pero el conjunto de oraciones que usan predicados de sustancia para el espacio-tiempo genera una ontología implícita que estructura todo el discurso.

Nota sobre el estatus epistémico. Formalmente, este teorema es elemental: demuestra que si un porcentaje es positivo, la lista filtrada no está vacía, y por tanto contiene al menos un elemento. Esto es una propiedad básica de las listas finitas. El valor del teorema no reside en la dificultad de la prueba, sino en el encuadre conceptual: formaliza la noción de "ontología fantasma" como una propiedad emergente del corpus que puede operacionalizarse y verificarse mecánicamente. La idea de que patrones globales del discurso implican instancias locales es conceptualmente productiva, aunque la demostración sea técnicamente sencilla.

¿Por qué este teorema no es trivial? Porque requiere:

13. Definir qué es una ontología fantasma (propiedad estadística del corpus).
14. Demostrar que si el porcentaje es positivo, la lista filtrada no está vacía.
15. Extraer un elemento concreto de la lista usando *cases*.
16. Razonamiento aritmético sobre desigualdades (usando *omega*).

4.4.4 Metateoremas estructurales del sistema [A-Meta]

Los teoremas anteriores muestran detecciones y propiedades emergentes. Ahora añadimos cinco metateoremas estructurales que caracterizan el comportamiento interno del sistema de tipos ontológicos.

Teorema T-DIST-ZERO [A-Meta]

Caracterización de la distancia ontológica nula.

Enunciado.

$$\forall (t_1, t_2 : \text{OntoType}),\ \text{ontological_distance}(t_1, t_2) = 0 \Leftrightarrow t_1 = t_2.$$

Demostración. Demostramos las dos implicaciones.

(⇒) Supongamos:

$$\text{ontological_distance}(t_1, t_2) = 0.$$

Analicemos los casos del pattern matching en la definición.

Si $(t_1, t_2) = (\text{relation}, \text{substance})$, el valor es $3 \neq 0$. Contradicción.

Si $(t_1, t_2) = (\text{process}, \text{substance})$, el valor es $2 \neq 0$. Contradicción.

Si $(t_1, t_2) = (\text{property}, \text{substance})$, el valor es $1 \neq 0$. Contradicción.

Si $(t_1, t_2) = (\text{formal_struct}, \text{substance})$, el valor es $1 \neq 0$. Contradicción.

Si $(t_1, t_2) = (\text{convention}, \text{substance})$, el valor es $2 \neq 0$. Contradicción.

En cualquier otro caso (catch-all), el valor es:

$$\text{if } t_1 = t_2 \text{ then } 0 \text{ else } 1.$$

Para que este valor sea 0, debe cumplirse $t_1 = t_2$.

Luego $t_1 = t_2$.

($\Leftarrow$) Supongamos $t_1 = t_2$. Entonces el par (t_1, t_1) no hace match con ninguno de los patrones específicos, pues todos ellos requieren que el segundo componente sea substance y que el primer componente pertenezca a:

$$\{\text{relation}, \text{process}, \text{property}, \text{formal_struct}, \text{convention}\}.$$

Como $t_1 = t_2$, si $t_1 = \text{substance}$, ningún patrón tiene primer componente substance; y si $t_1 \neq \text{substance}$, el segundo componente es $t_1 \neq \text{substance}$. En ambos subcasos, el par cae en el catch-all, donde:

$$\text{if } t_1 = t_1 \text{ then } 0 \text{ else } 1 = 0.$$

Luego:

$$\text{ontological_distance}(t_1, t_2) = 0.$$

▫

Teorema T-DIST-ID [A-Meta]

La distancia ontológica de un tipo consigo mismo es nula.

Enunciado.

$$\forall(t\colon \text{OntoType}),\ \text{ontological_distance}(t, t) = 0.$$

Demostración. Sea $t \in \text{OntoType}$. Consideremos el par (t, t) en la definición de ontological_distance.

Si $t = \text{relation}$, el patrón $(\text{relation}, \text{substance})$ requiere segundo componente substance $\neq$ relation. Luego no hace match. Los demás patrones específicos fallan por el primer componente. El par cae en catch-all, produciendo:

$$\text{if relation} = \text{relation then 0 else } 1 = 0.$$

Si $t = \text{process}$, análogamente el segundo componente es process ≠ substance, luego ningún patrón específico hace match. El catch-all produce 0.

Si $t = \text{property}$, el segundo componente es property ≠ substance. Ningún patrón específico aplica. Catch-all: 0.

Si $t = \text{formal_struct}$, el segundo componente es formal_struct ≠ substance. Ningún patrón específico aplica. Catch-all: 0.

Si $t = \text{convention}$, el segundo componente es convention ≠ substance. Ningún patrón específico aplica. Catch-all: 0.

Si $t = \text{substance}$, ningún patrón específico tiene primer componente substance. El par cae en catch-all, produciendo:

$$\text{if substance} = \text{substance then 0 else } 1 = 0.$$

En todos los casos:

$$\text{ontological_distance}(t, t) = 0.$$

▫

Teorema T-INC-SYM [A-Meta]

Simetría de la incompatibilidad entre atributos.

Enunciado.

$\forall(p_1, p_2 : \text{AttributedProperty}),\ \text{are_incompatible}(p_1, p_2) \Leftrightarrow \text{are_incompatible}(p_2, p_1)$.

Demostración. Sean $p_1, p_2 \in$ AttributedProperty. Por definición:

$\text{are_incompatible}(p_1, p_2) \Leftrightarrow (p_1 = \text{has_psychology} \wedge p_2 = \text{no_psychology}) \vee (p_1 = \text{no_psychology} \wedge p_2 = \text{has_psychology})$.

Por conmutatividad de la conjunción lógica (∧), la primera disyunción es lógicamente equivalente a:

$$(p_2 = \text{no_psychology} \wedge p_1 = \text{has_psychology}),$$

y la segunda disyunción es equivalente a:

$$(p_2 = \text{has_psychology} \wedge p_1 = \text{no_psychology}).$$

Reordenando:

$\text{are_incompatible}(p_1, p_2) \Leftrightarrow (p_2 = \text{has_psychology} \wedge p_1 = \text{no_psychology}) \vee (p_2 = \text{no_psychology} \wedge p_1 = \text{has_psychology})$.

El lado derecho es exactamente are_incompatible(p_2, p_1) por definición. ▫

Teorema T-INC-IRR [A-Meta]

Irreflexividad de la incompatibilidad.

Enunciado.

$$\forall(p\text{: AttributedProperty}),\ \neg\text{are_incompatible}(p, p).$$

Demostración. Sea $p \in$ AttributedProperty. Por definición:

$$\text{are_incompatible}(p, p) \Leftrightarrow (p = \text{has_psychology} \wedge p = \text{no_psychology}) \vee (p = \text{no_psychology} \wedge p = \text{has_psychology}).$$

Supongamos, para reducir al absurdo, que la primera disyunción es verdadera. Entonces:

$$p = \text{has_psychology}$$

y

$$p = \text{no_psychology}.$$

Luego:

$$\text{has_psychology} = \text{no_psychology}.$$

Pero estos son constructores distintos de un tipo inductivo, luego son distintos. Contradicción.

Supongamos que la segunda disyunción es verdadera. Entonces:

$$p = \text{no_psychology}$$

y

$$p = \text{has_psychology}.$$

Luego:

$$\text{no_psychology} = \text{has_psychology}.$$

Misma contradicción.

Como ambas disyunciones son falsas, la disyunción completa es falsa. Luego:

$$\neg\text{are_incompatible}(p, p).$$

▫

Teorema T-BOOL-REF [A-Meta]

Equivalencia entre la versión booleana y la proposicional de la reificación.

Enunciado.

$$\forall(e\text{: PhysicsEntity}), \forall(p\text{: PhysicsPredicate}),\ \text{is_reified_bool}(e,p) = \text{true} \Leftrightarrow \text{is_grammatical_reification}(e,p).$$

Demostración. Sean e y p arbitrarios. Recordemos las definiciones:

$$\text{is_reified_bool}(e,p) = \text{if ontoType}(e) = \text{requiredSubjectType}(p) \text{ then false else true,}$$

$$\text{is_grammatical_reification}(e,p) \Leftrightarrow \text{ontoType}(e) \neq \text{requiredSubjectType}(p).$$

Distinguimos dos casos exhaustivos.

Caso 1: $\text{ontoType}(e) = \text{requiredSubjectType}(p)$. Entonces:

$$\text{is_reified_bool}(e,p) = \text{false}.$$

Y:

$$\text{is_grammatical_reification}(e,p) = (\text{ontoType}(e) \neq \text{requiredSubjectType}(p)) = \text{false}.$$

Luego ambos lados de la equivalencia son falsos, y la equivalencia se satisface.

Caso 2: $\text{ontoType}(e) \neq \text{requiredSubjectType}(p)$. Entonces:

$$\text{is_reified_bool}(e,p) = \text{true}.$$

Y:

$$\text{is_grammatical_reification}(e,p) = \text{true}.$$

Luego ambos lados son verdaderos, y la equivalencia se satisface.

En ambos casos la equivalencia se verifica. ▫

4.5 Los 7 procesos satélite como operadores gramaticales [A/C]

En el Capítulo 1 (sección 1.2.1), nosotros identificamos siete procesos que estabilizan la reificación: idealización, personificación, naturalización, totalización, teleologización, sacralización y temporalización. En la sección 4.4, nosotros formalizamos tres de ellos (personificación, naturalización, sacralización) como estructuras que atribuyen propiedades. Aquí mostramos cómo operan en el discurso físico.

Proceso	Ejemplo en física	Operación gramatical
Naturalización	“Las leyes de la naturaleza” como	Presenta una convención

	entidades eternas e intocables, independientes del marco teórico que las formula.	(el marco teórico) como si fuera un dato natural.
Personificación	"La naturaleza aborrece el vacío", "la naturaleza no salta", "el universo busca el equilibrio".	Atribuye psicología (aborrecer, buscar) a un proceso.
Sacralización	"La ecuación más bella de la física", "la elegancia de la teoría", presentación de ciertas ecuaciones como intocables.	Eleva una estructura formal a un estatus que exige reverencia y bloquea la revisión.
Teleologización	"El universo tiende al equilibrio", "la entropía siempre aumenta" presentado como dirección necesaria y no como regularidad estadística.	Atribuye dirección y propósito a un proceso.
Idealización	"El modelo estándar es la teoría final", "la teoría del todo".	Eleva una construcción a un ideal de completitud y perfección.
Totalización	"La física explica todo", "la teoría unificada".	Presenta un dominio parcial como si cubriera la totalidad.
Temporalización	"La flecha del tiempo" como propiedad del universo y no como asimetría termodinámica.	Convierte una regularidad estadística en una entidad permanente que "avanza".

Nota sobre el estatus epistémico. La formalización de los procesos satélite como estructuras con atributos es [A] (compila en Lean 4). La identificación de ejemplos específicos en el discurso físico es [C] (interpretación basada en análisis del discurso). De los siete procesos identificados en el Capítulo 1, nosotros formalizamos tres (personificación, naturalización, sacralización) como ilustración del patrón. Los cuatro restantes (idealización, totalización, teleologización, temporalización) se presentan solo como análisis discursivo [C], sin formalización en Lean. La extensión a los siete procesos es conceptualmente directa pero requeriría definir propiedades adicionales para cada proceso.

4.6 Evidencia empírica: Edelstein, Maldacena, Deser [B/C]

La entrevista a José Edelstein

En la entrevista del 27 de julio de 2026 en el programa "El Maravilloso Mundo de la Física" (Gelatina), el físico teórico José Edelstein proporcionó ejemplos directamente observables de reificación en el discurso de divulgación [B]. Nosotros identificamos los siguientes:

Ejemplo 1: "Nos llegan vibraciones del espacio-tiempo"

Edelstein dice: "No solo nos llega luz del cielo, nos llegan vibraciones del espacio-tiempo, nos llegan partículas elementales en cantidades industriales."

Análisis gramatical:

- Sujeto: "vibraciones del espacio-tiempo".
- El espacio-tiempo aparece como un medio material que "vibra".
- Tipo atribuido: substance (algo que puede vibrar).
- Tipo real: formal_struct (variedad con métrica).
- Veredicto: reificación gramatical ✓.
- Proceso satélite: naturalización (presenta la métrica como un medio físico natural).

Ejemplo 2: "El universo es un gran mecanismo de relojería"

Edelstein dice, refiriéndose al universo newtoniano: "El universo newtoniano tiene leyes deterministas y mecanicistas. El universo es un gran mecanismo de relojería en el cual todo se mueve porque una causa previa hizo que se mueva inexorablemente de la manera que lo hace."

Análisis gramatical:

- Sujeto: "el universo".
- Predicado: "es un gran mecanismo de relojería".
- El universo aparece como un objeto mecánico (sustancia).
- Tipo atribuido: substance (mecanismo).
- Tipo real: process (conjunto de procesos físicos).
- Veredicto: reificación gramatical ✓.
- Proceso satélite: naturalización + totalización (presenta la totalidad de los procesos como un objeto mecánico unitario).

Nota importante. Edelstein mismo reconoce la distinción entre el mapa y el territorio en la misma entrevista. Cita a Borges y dice: "El éxito de la ciencia justamente radica en el éxito de los mapas. Si el mapa necesitara detallar hasta la farola y el perrito que está paseando, sería un mapa irrelevante." Esto muestra que los físicos teóricos saben que el formalismo no es la realidad. La reificación no ocurre por ignorancia; ocurre como prótesis pedagógica necesaria para comunicar estructuras abstractas en lenguaje natural. El problema surge cuando la audiencia (y a veces el propio físico en sus horas de filosofía) confunde la prótesis con la realidad.

Ejemplo 3: "Ni el vaso sabe si está lleno o vacío"

Edelstein dice, sobre el mundo cuántico: "El no saber dónde está o el no saber si está lleno no es porque ignorancia mía, es que ni el vaso sabe si está lleno o vacío."

Análisis gramatical:

- Sujeto: “el vaso” (en contexto cuántico).
- Predicado: “sabe”.
- Se atribuye un estado epistémico (“saber”) a un sistema físico.
- Tipo atribuido: substance con psicología (agente que “sabe”).
- Tipo real: process (sistema cuántico en superposición).
- Veredicto: personificación ✓.

La conferencia de Maldacena

En la conferencia del 28 de julio de 2026 en el Planetario de Buenos Aires (Usina del Arte), Juan Martín Maldacena presentó la charla “El significado del espacio-tiempo: agujeros negros y entrelazamiento cuántico” [B]. La sinopsis oficial plantea las preguntas: “¿Qué son el espacio y el tiempo? ¿Qué ocurre en el interior de un agujero negro? ¿Qué relación existe entre la gravedad y la mecánica cuántica?”

La relevancia para este capítulo es la siguiente: la correspondencia AdS/CFT (1997) propone que la geometría del espacio-tiempo en el “bulk” (interior) es dual a una teoría cuántica de campos en la frontera. Si esta conjetura es correcta, el espacio-tiempo no es el nivel fundamental de descripción; es una estructura emergente derivada del entrelazamiento cuántico. Esto no “refuta” la relatividad general, pero muestra que la geometría puede no ser una sustancia primaria sino una estructura derivada de relaciones cuánticas.

Nota sobre el estatus epistémico de AdS/CFT. La correspondencia AdS/CFT es una conjetura con evidencia matemática considerable, pero no ha sido confirmada experimentalmente de manera directa. Nosotros la citamos como evidencia de que el discurso físico de frontera contempla formulaciones no sustancialistas, no como prueba de que “el espacio-tiempo no existe”. El estatus epistémico es: conjetura con soporte matemático fuerte, no teoría confirmada experimentalmente.

La derivación de Deser (1970)

Stanley Deser demostró en 1970 que las ecuaciones de Einstein pueden derivarse partiendo de un campo de espín-2 en espacio plano de Minkowski, sin postular curvatura geométrica [B]. La derivación procede por autoconsistencia: se comienza con un campo $h_{\mu\nu}$ en espacio plano, se exige que el campo sea fuente de sí mismo (acoplamiento universal), y la iteración converge a las ecuaciones de Einstein.

La relevancia para este capítulo: la “geometría curva” no es la única formulación posible de la relatividad general. Es un reempaquetamiento matemático que resulta elegante y poderoso, pero que no es ontológicamente necesario. El formalismo puede expresarse sin geometría curva. Esto refuerza la tesis de que la reificación geométrica (“el espacio-tiempo se curva”) es una elección de representación, no una descripción de una sustancia que se deforma.

4.7 La pedagogía de la reificación [B/C]

La investigación en educación física (PER) demuestra que la reificación no es un error accidental en la enseñanza de la física; es una estrategia pedagógica explícita y deliberada [B].

La reificación como herramienta heurística

La investigación muestra que:

- Hablar de "calor que fluye" permite a los estudiantes aplicar su intuición sobre fluidos a un fenómeno térmico [B].
- Personificar la energía como algo que se "almacena" o se "usa" hace accesible un concepto abstracto [B].
- Los modelos pedagógicos (la sábana elástica, el mecanismo de relojería) son reificaciones controladas que sirven como anclajes cognitivos [B].

La reificación implícita como fuente de misconceptions

En la investigación en enseñanza de la física (PER, por sus siglas en inglés), el término misconceptions —que aquí traducimos como concepciones erróneas— no designa simples despistes o malentendidos pasajeros, sino ideas profundamente arraigadas que los estudiantes construyen para dar sentido al mundo y que resultan incompatibles con el conocimiento científico aceptado. A diferencia de un error puntual por falta de información, una concepción errónea es sistemática, internamente coherente desde la lógica cotidiana y sorprendentemente resistente a la instrucción formal. Por ejemplo, creer que "la energía se gasta" o que "la aceleración siempre va en dirección del movimiento" no es un olvido, sino la manifestación de una ontología implícita en la que la energía se trata como una sustancia consumible o el movimiento como una propiedad que los objetos poseen. Estas concepciones erróneas suelen originarse precisamente en las metáforas reificantes que impregnan el discurso pedagógico; cuando la analogía de la sábana elástica deja de ser una prótesis cognitiva consciente y se convierte en una descripción literal del espacio-tiempo, la reificación gramatical se fosiliza en un error ontológico persistente. Nuestro sistema formal de tipos ontológicos detecta exactamente ese desajuste: la atribución a una entidad de propiedades que no corresponden a su tipo real.

Cuando la reificación se realiza sin advertencia explícita, produce errores ontológicos persistentes en los estudiantes [B]:

- "La aceleración siempre va en dirección del movimiento" (confusión entre proceso y estado).
- "La inercia es una fuerza" (confusión entre propiedad y relación).
- "La energía se gasta" (tratamiento de una propiedad como sustancia consumible).

La conexión con nuestro sistema formal

Lo que la investigación en PER identifica como "ontological misconceptions" es exactamente lo que nuestro sistema formal detecta como `is_grammatical_reification`. La PER lo describe en prosa; nosotros lo formalizamos como un mismatch de tipos verificable en Lean 4.

La PER propone como solución desarrollar "conciencia ontológica" en los estudiantes: enseñarles qué tipo de cosa es cada concepto antes de enseñarles sus propiedades [B]. Nuestro sistema formal es una implementación verificable de esa propuesta: clasifica entidades por tipo ontológico y detecta cuándo se les atribuyen propiedades que no corresponden a su tipo.

La paradoja pedagógica

La investigación revela una paradoja [C]:

17. La reificación es necesaria para el aprendizaje inicial (sin ella, los conceptos abstractos son inaccesibles).
18. La reificación es perniciosa si no se desactiva posteriormente (produce misconceptions persistentes).

La solución que la PER propone es la misma que nosotros proponemos en este libro: usar la reificación como prótesis cognitiva, pero hacer visible la prótesis. Enseñar con la metáfora, pero señalar que es una metáfora. Usar la sábana elástica, pero decir "esto es una analogía; el formalismo real es una variedad con métrica".

Esto conecta directamente con la analogía del espejismo que desarrollamos en el Capítulo 1 (sección 1.2.2): la reificación es como el espejismo del desierto. No es un error del observador; es una consecuencia inevitable de la física atmosférica (o de la gramática del lenguaje natural). Pero el observador puede aprender a reconocer el espejismo. La "conciencia ontológica" es el aprendizaje de reconocer el espejismo.

4.8 Limitaciones: qué puede y qué no puede detectar este sistema [C]

Para completar el análisis, nosotros señalamos explícitamente lo que este sistema no puede hacer.

Primera limitación: la clasificación ontológica es una elección, no un descubrimiento.

La función `ontoType` clasifica el espacio-tiempo como `.formal_struct` y la energía como `.property`. Esta clasificación se ancla en el formalismo físico estándar, pero no es la única posible. Una ontología de "ontología primitiva de la energía" clasificaría la energía como `.substance`. Una ontología procesual (Whitehead) clasificaría la materia

como `.process`. Si el lector cambia la clasificación, los teoremas cambian. Esto no es un defecto; es una consecuencia de lo que establecimos en el Capítulo 2 (sección 2.7): los axiomas son elecciones filosóficas.

Segunda limitación: el sistema detecta mismatches de tipo, no intenciones.

Cuando un físico dice "el espacio-tiempo se curva" como metáfora pedagógica consciente, el sistema detecta el mismatch de tipo. Pero no distingue entre:

19. Un físico que sabe que es una metáfora y la usa deliberadamente (prótesis cognitiva).
20. Un divulgador que cree literalmente que el espacio-tiempo es una tela elástica (reificación inconsciente).
21. Un texto de divulgación que no explicita la metáfora (reificación implícita).

El sistema detecta la estructura gramatical de la reificación. La interpretación de la intención requiere análisis del discurso adicional.

Tercera limitación: el sistema es finito y estático.

El sistema clasifica siete entidades y ocho predicados. El discurso físico real contiene miles de entidades y predicados. El sistema es un modelo mínimo que ilustra la estructura, no una base de datos completa. Extenderlo requeriría añadir más entidades y predicados, lo cual es conceptualmente directo pero laborioso.

Cuarta limitación: el sistema no modela la dinámica de la reificación.

La reificación no es un evento puntual; es un proceso que se estabiliza con el tiempo mediante los siete procesos satélite. El sistema actual detecta el mismatch en un instante dado, pero no modela cómo la reificación se construye, se estabiliza y se desactiva a lo largo del tiempo. Modelar esa dinámica requeriría un sistema temporal o de eventos, que es trabajo futuro.

Quinta limitación: el sistema no resuelve el problema pedagógico.

Detectar que "el espacio-tiempo se curva" es una reificación gramatical no dice cómo enseñar relatividad general sin esa metáfora. La solución pedagógica (usar la metáfora pero señalarla como metáfora) requiere una intervención didáctica que el sistema formal no puede proporcionar. El sistema diagnostica; la intervención es tarea del pedagogo.

4.9 Conexión con los capítulos anteriores y siguientes

Conexión con el Capítulo 1 (reificación como error categorial)

El sistema formalizado en este capítulo es una implementación verificable del protocolo de cuatro preguntas del Capítulo 1, aplicada específicamente al discurso

físico. La Pregunta 1 (tipo del objeto) se implementa como *ontoType*. La Pregunta 2 (atribución indebida) se implementa como *requiredSubjectType*. La Pregunta 3 (reformulación alternativa) se ilustra con la derivación de Deser y la correspondencia AdS/CFT. La Pregunta 4 (teoría resuelta con narrativa sucia) se aplica al caso del espacio-tiempo.

Conexión con el Capítulo 3 (estratificación como anti-reificación)

En el Capítulo 3, la estratificación impide que el predicado de verdad se aplique a sí mismo. En este capítulo, la clasificación por tipos ontológicos impide que un predicado de sustancia se aplique a una estructura relacional. El principio es el mismo: restricción de tipo como mecanismo anti-reificatorio.

Diferencia importante: En SPA, la restricción es preventiva: el compilador impide construir fórmulas mal formadas. En este capítulo, la restricción es diagnóstica: el sistema detecta reificaciones después de que han sido formuladas en lenguaje natural. La diferencia no es trivial: SPA bloquea la reificación antes de que ocurra; este sistema la detecta después. Ambos son mecanismos anti-reificatorios, pero operan en momentos distintos del proceso discursivo.

Conexión con el Capítulo 5 (medición cuántica)

En el Capítulo 5, nosotros aplicaremos el mismo protocolo al caso de la medición cuántica: el "colapso de la función de onda" como reificación de una actualización epistémica (cambio de información) tratada como evento físico (sustancia que "colapsa"). El sistema formal de este capítulo puede extenderse para incluir entidades cuánticas (función de onda, observable, estado entrelazado) y predicados cuánticos (colapsa, se entrelaza, se superpone).

Conexión con el Capítulo 6 (sistemas normativos)

En el Capítulo 6, nosotros aplicaremos el protocolo al discurso jurídico y político. La estructura es análoga: "el Estado decide" reifica una construcción institucional (relación entre personas y normas) tratándola como agente unitario (sustancia con voluntad). El sistema SFCD formalizado en Z3 (Capítulo 2) es el análogo normativo del sistema de tipos ontológicos que presentamos aquí.

4.10 Síntesis de demostraciones formales del capítulo [A-Meta]

Nosotros presentamos la siguiente tabla de demostraciones formales en prosa integradas en este capítulo. Estas pruebas elevan el estatus epistémico de las secciones formales de [A-Código] a [A-Meta] + [A-Código], replicando el protocolo usado en el Capítulo 5.

Designación	Enunciado	Método	Sección
T-REF-1	"El espacio-tiempo se curva" es	Inspección de tipos	4.3

	reificación		
T-REF-2	“La energía fluye” es reificación	Inspección de tipos	4.3
T-REF-3	“La naturaleza aborrece el vacío” es reificación	Inspección de tipos	4.3
T-REF-4	“La energía se almacena” es reificación	Inspección de tipos	4.3
T-REF-5	“El espacio-tiempo vibra” es reificación	Inspección de tipos	4.3
T-NOREF-1	“El campo métrico relaciona eventos” no es reificación	Igualdad de tipos	4.3
T-NOREF-2	“La estructura tiene la propiedad de curvatura” no es reificación	Igualdad de tipos	4.3
T-INCOMP	Personificación y naturalización son incompatibles	Exhibición de testigos	4.4
T-DIST-GE1	Distancia a sustancia positiva para tipos no sustanciales	Análisis de casos	4.4
T-DIST-SUB	Procesos que apuntan a sustancia aumentan distancia	Consecuencia de T-DIST-GE1	4.4
T-PHANTOM	Ontología fantasma implica reificación	Lista filtrada no vacía	4.4
T-DIST-ZERO	Distancia nula si y solo si igualdad de tipos	Análisis de pattern matching	4.4.4
T-DIST-ID	Distancia de un tipo consigo mismo es nula	Análisis de casos	4.4.4
T-INC-SYM	Simetría de la incompatibilidad	Conmutatividad lógica	4.4.4
T-INC-IRR	Irreflexividad de la incompatibilidad	Reducción al absurdo	4.4.4
T-BOOL-REF	Equivalencia entre versión booleana y proposicional	Análisis de casos	4.4.4

Conclusión metodológica [C]. El Capítulo 4 posee materia prima para 16 demostraciones formales en prosa. Estas pruebas son correctas, completas y verificables por un lector con formación básica en lógica proposicional, teoría de tipos elemental y semántica de lenguajes funcionales. Su integración junto a los códigos Lean eleva el estatus epistémico del capítulo de una verificación meramente computacional a una demostración formal mixta **[A-Meta] + [A-Código]**.

Etiquetado del capítulo

Sección	Contenido	Clasificación

4.1	Aplicación del protocolo al espacio-tiempo	[C] Interpretación
4.2	Sistema de tipos ontológicos en Lean 4	[A-Código]
4.3	Teoremas básicos de detección de reificación	[A-Meta] + [A-Código]
4.4	Teoremas no triviales (incompatibilidad, acumulación, emergencia)	[A-Meta] + [A-Código]
4.4.4	Metateoremas estructurales del sistema	[A-Meta]
4.5	Procesos satélite como operadores (estructura formal)	[A-Código]
4.5	Identificación de ejemplos en física	[C] Interpretación
4.6	Entrevista a Edelstein	[B] Citada de fuente
4.6	Conferencia de Maldacena	[B] Citada de fuente
4.6	Derivación de Deser	[B] Citada de literatura
4.7	Pedagogía de la reificación (PER)	[B] Citada de literatura
4.7	Conexión con nuestro sistema	[C] Interpretación
4.8	Limitaciones del sistema	[C] Interpretación
4.9	Conexiones con otros capítulos	[C] Interpretación
4.10	Síntesis de demostraciones formales	[A-Meta] + [C]

Capítulo 5: La medición cuántica como cambio de nivel

5.0 Introducción: el problema de la medición cuántica

En el Capítulo 4, nosotros formalizamos la gramática de la reificación en el discurso físico clásico, mostrando que el espacio-tiempo, la energía y la fuerza son tratados como sustancias cuando el formalismo los define como estructuras relacionales. En este capítulo, aplicamos el mismo protocolo a un caso paradigmático de la física moderna: la medición cuántica y el llamado "colapso de la función de onda".

La narrativa divulgativa estándar presenta el colapso como un evento físico: una función de onda que "colapsa" abruptamente desde una superposición de estados a un estado definido cuando se realiza una medición. Esta narrativa sugiere que algo cambia en el sistema físico mismo: la función de onda, tratada como una entidad sustancial, sufre una transformación abrupta [B].

Hay que ser cuidadosos aquí: esta narrativa es en parte una caricatura didáctica, pero como demostramos en los capítulos anteriores, la reificación no es simplemente un error pedagógico; está profundamente facilitada por la estructura gramatical del lenguaje mismo. El lenguaje natural, con su estructura sujeto-predicado, nos empuja constantemente a tratar procesos y relaciones como si fueran sustancias con propiedades. Cuando decimos "la función de onda colapsa", la gramática exige un sujeto sustancial ("la función de onda") que realiza una acción ("colapsa"), aunque el formalismo matemático describa simplemente una actualización de distribuciones de probabilidad. La reificación cuántica es, por tanto, un caso paradigmático de cómo la gramática del lenguaje natural conspira contra la precisión ontológica del formalismo matemático.

La interpretación de Copenhague, en las formulaciones originales de Bohr y Heisenberg, ya enfatizaba el carácter epistémico de la función de onda. Bohr escribió: "No hay un mundo cuántico. Solo hay una descripción cuántica abstracta de la realidad" [B]. La reificación del colapso es más un producto de la pedagogía simplificada y de las constricciones gramaticales del lenguaje natural que de la interpretación original. Sin embargo, la narrativa sustancialista sigue siendo dominante en libros de texto y divulgación, y produce misconceptions persistentes [B].

Existe una narrativa alternativa, respaldada por interpretaciones como QBism de Fuchs, la Mecánica Cuántica Operacional (OQM) de Werner, y lecturas epistémicas de Copenhague: el "colapso" no es un evento físico en el sistema, sino una actualización epistémica en el observador. Lo que cambia no es el sistema, sino la información disponible sobre el sistema. La función de onda no es una sustancia que colapsa; es una representación de nuestro conocimiento que se actualiza [B].

Estructura formal del capítulo: física, lógica y dinámica discursiva

Este capítulo emplea una estrategia formal tripartita que es importante anunciar desde el inicio para orientar al lector. La reificación en la mecánica cuántica tiene una característica peculiar que la distingue de los capítulos anteriores: la reificación no se produce en los objetos matemáticos mismos, sino en el paso de su interpretación. El vector de estado en un espacio de Hilbert no está reificado; la reificación aparece cuando se interpreta ese vector como "una cosa que colapsa".

Por ello, el capítulo procede en tres niveles de formalización:

- **Lean 4 (Secciones 5.2–5.3):** Formalizamos las propiedades estructurales genuinas de la mecánica cuántica: no-conmutatividad, violación de las desigualdades de Bell, contextualidad y bloqueo de la definición contracausal. Estos teoremas establecen los límites algebraicos y analíticos que distinguen el comportamiento cuántico del clásico. Aquí demostramos física, no reificación. Las demostraciones en prosa **[A-Meta]** acompañan a las verificaciones formales **[A-Código]**.

- **Z3 (Sección 5.4.1):** Formalizamos el mapa del pluralismo interpretativo. Demostramos que el teorema de Bell no refuta "el realismo" en abstracto, sino una conjunción específica de tres supuestos metafísicos (Localidad, Definición Contracausal, Independencia Estadística). Z3 mapea exhaustivamente las 7 interpretaciones viables que surgen al renunciar a al menos uno de estos supuestos, mostrando que la interpretación epistémica es una opción entre varias, no una conclusión lógicamente inevitable.

- **Clingo (Sección 5.4.2):** Modelamos la dinámica histórica del discurso cuántico. Simulamos cómo una comunidad científica transita entre las 7 posibilidades del mapa de Z3 cuando llega evidencia contraria al paradigma dominante. El resultado revela que la interpretación epistémica no es la trayectoria óptima bajo criterios de parsimonia pura; su adopción requiere argumentos filosóficos adicionales.

Esta estructura permite al lector comprender primero qué dice la física (Lean), luego qué espacio de interpretaciones es lógicamente viable (Z3), y finalmente cómo evoluciona el debate científico (Clingo). Las secciones de Lean son las más técnicas; el lector puede saltarlas y centrarse en las interpretaciones prosaicas, que son autosuficientes.

Nota sobre el alcance de la formalización. Nosotros no formalizamos la mecánica cuántica completa (espacios de Hilbert de dimensión infinita, operadores autoadjuntos, evolución unitaria). Lo que nosotros formalizamos es un modelo estructural con qubits y ángulos de medición que captura propiedades esenciales de la teoría cuántica. El modelo es didáctico pero no trivial: los teoremas demuestran límites algebraicos y analíticos verificables que distinguen el comportamiento epistémico cuántico de cualquier teoría clásica de variables ocultas locales.

5.1 El caso del colapso de la función de onda: aplicación del protocolo [C]

Aplicamos las cuatro preguntas del Capítulo 0 al caso del colapso de la función de onda, manteniendo la neutralidad interpretativa: el protocolo analiza el formalismo matemático sin presuponer ninguna interpretación particular.

Pregunta 1: ¿Cuál es el objeto formal y cuál es su tipo?

El objeto formal es la función de onda ψ, que en la formulación estándar es un vector unitario en un espacio de Hilbert. Matemáticamente, codifica una distribución de probabilidad sobre los posibles resultados de medición, calculada mediante la regla de Born: $P(a_i) = |\langle a_i|\psi\rangle|^2$. El formalismo por sí mismo es agnóstico respecto al estatus ontológico de este objeto: podría ser una entidad física real (interpretación ontológica) o un recurso predictivo (interpretación epistémica).

En la formulación estándar, la función de onda evoluciona de dos maneras:

- **Evolución unitaria (ecuación de Schrödinger):** determinista, continua, reversible.
- **Colapso (postulado de medición):** probabilístico, discontinuo, irreversible.

Esta dualidad es la fuente del "problema de la medición": ¿por qué hay dos reglas de evolución distintas?

Pregunta 2: ¿Qué atribuye la narrativa a este objeto que el formalismo no exige?

La narrativa divulgativa atribuye a la función de onda propiedades de sustancia física:

22. "La función de onda colapsa" (como si fuera un objeto que sufre una transformación física).
23. "El electrón está en una superposición de estados" (como si la superposición fuera un estado físico real, no una representación matemática).
24. "La medición fuerza al sistema a elegir un estado" (como si el sistema tuviera una ontología definida que la medición revela).

Estas atribuciones requieren un sujeto de tipo sustancia (algo que puede colapsar, estar en superposición, ser forzado). El formalismo matemático no exige estas atribuciones; son una capa interpretativa añadida. Como demostramos en el Capítulo 1, la estructura gramatical del lenguaje natural (sujeto-predicado) facilita esta reificación: el lenguaje nos empuja a tratar "la función de onda" como un sustantivo que denota una entidad, cuando en el formalismo es simplemente un vector en un espacio abstracto.

Pregunta 3: ¿Existen descripciones alternativas?

Sí, y son múltiples. Hay al menos dos familias de interpretaciones:

25. **Interpretaciones ontológicas:** tratan la función de onda como entidad física real (teoría de Bohm, muchos mundos, colapso objetivo).
26. **Interpretaciones epistémicas:** tratan la función de onda como representación de información (QBism, OQM, Copenhague estricto).

Bajo las interpretaciones epistémicas, el "colapso" se trata como una actualización bayesiana: el observador actualiza su información sobre el sistema tras obtener un resultado de medición. El sistema físico no cambia; lo que cambia es la información del observador.

Pregunta 4: ¿Es teoría resuelta con narrativa sucia?

Parcialmente. La mecánica cuántica es una teoría resuelta en el sentido de que las predicciones empíricas están confirmadas con precisión extraordinaria (violación de desigualdades de Bell, efecto Casimir, computación cuántica). Pero la interpretación de la teoría sigue siendo objeto de debate.

En las secciones siguientes, nosotros argumentaremos que la interpretación epistémica es la opción más parsimoniosa porque evita postular eventos físicos misteriosos (colapsos instantáneos, acción a distancia), aunque a costa de renunciar al realismo clásico. Esta elección no es una derivación necesaria del protocolo, sino una hipótesis interpretativa que defendemos por su coherencia con el principio anti-reificador que articula este trabajo.

5.2 Formalización del modelo epistémico en Lean 4 [A-Código]

Nosotros construimos un sistema formal en Lean 4 que modela la medición cuántica desde una perspectiva puramente epistémica. Aquí no postulamos un estado ontológico subyacente con valores definidos. La superposición, la no-conmutatividad y el entrelazamiento se modelan enteramente como propiedades de los estados de creencia del agente.

Definiciones operacionales: agente, información y frontera

Agente. En nuestro modelo, un "agente" es cualquier sistema macroscópico capaz de registrar un resultado de medición de manera termodinámicamente irreversible. No requiere conciencia; puede ser un aparato de medición, un computador o un observador humano.

Información. La "información" se entiende aquí como recurso predictivo operacional, no como creencia subjetiva (QBism) ni como conocimiento objetivo (Copenhague). Es el conjunto de probabilidades que un agente racional debe asignar a futuros resultados de medición.

Frontera. El "problema de la frontera" pregunta dónde termina el sistema cuántico y dónde empieza el agente clásico. Nosotros adoptamos un criterio pragmático: la frontera es el umbral de decoherencia, la interacción irreversible con el entorno macroscópico que registra un resultado de manera estable.

Código Lean 4: tipos, medición y correlaciones

```
import Mathlib.Tactic
import Mathlib.Analysis.SpecialFunctions.Trigonometric.Basic

open Real

noncomputable section

-- Ejes de medición (observables incompatibles)
inductive Axis
  | Z | X | D | D'
  deriving DecidableEq, Repr

-- Resultados de medición
inductive SpinResult
  | up | down
  deriving DecidableEq, Repr

-- Función auxiliar: mapeo de SpinResult a enteros ±1
def spin_to_int (s : SpinResult) : Int :=
  match s with
  | .up => 1
  | .down => -1

theorem spin_to_int_cases (s : SpinResult) : spin_to_int s = 1 ∨
spin_to_int s = -1 := by
  cases s <;> simp [spin_to_int]

-- Estado epistémico: creencias del agente
structure EpistemicState where
  prob_up_Z : Nat
  prob_up_X : Nat
  prob_up_D : Nat
  prob_up_D' : Nat
  deriving DecidableEq, Repr

-- Medición como actualización epistémica
def measure_Z (_prior : EpistemicState) (result : SpinResult) :
EpistemicState :=
  match result with
  | .up => ⟨100, 50, 50, 50⟩
  | .down => ⟨0, 50, 50, 50⟩

def measure_X (_prior : EpistemicState) (result : SpinResult) :
```

```
EpistemicState :=
  match result with
  | .up => ⟨50, 100, 50, 50⟩
  | .down => ⟨50, 0, 50, 50⟩

def measure_D (_prior : EpistemicState) (result : SpinResult) :
EpistemicState :=
  match result with
  | .up => ⟨50, 50, 100, 50⟩
  | .down => ⟨50, 50, 0, 50⟩

-- Estados preparados
def prepared_Z_up : EpistemicState := ⟨100, 50, 50, 50⟩
def prepared_X_up : EpistemicState := ⟨50, 100, 50, 50⟩
def prepared_maximally_mixed : EpistemicState := ⟨50, 50, 50, 50⟩

-- Correlación cuántica: E(θa, θb) = -cos(θa - θb)
def quantum_correlation (θa θb : ℝ) : ℝ := -cos (θa - θb)

def angle_a  : ℝ := 0
def angle_a' : ℝ := π/2
def angle_b  : ℝ := π/4
def angle_b' : ℝ := 3*π/4

def S_quantum : ℝ :=
  quantum_correlation angle_a angle_b - quantum_correlation angle_a
angle_b' +
  quantum_correlation angle_a' angle_b + quantum_correlation angle_a'
angle_b'

-- Correlación clásica: producto de resultados ±1
def classical_correlation (assignA assignB : Axis → SpinResult) (a b :
Axis) : Int :=
  (spin_to_int (assignA a)) * (spin_to_int (assignB b)) * 100

def classical_S (assignA assignB : Axis → SpinResult) : Int :=
  classical_correlation assignA assignB .Z .D -
  classical_correlation assignA assignB .Z .D' +
  classical_correlation assignA assignB .X .D +
  classical_correlation assignA assignB .X .D'

end
```

5.3 Teoremas estructurales de la mecánica cuántica en Lean 4 [A-Meta] + [A-Código]

Nota sobre la accesibilidad del código. Las demostraciones formales que siguen utilizan tácticas de Lean 4 y Mathlib que pueden ser técnicas para lectores sin formación en asistentes de prueba. El lector puede saltar las pruebas formales y centrarse en las interpretaciones prosaicas, que son autosuficientes.

Nota sobre el estatus epistémico. Los Teoremas 1, 3 y 5 son verificaciones de consistencia estructural (sanity checks) que garantizan que nuestro modelo respeta sus propios axiomas de diseño. El peso físico-filosófico sustantivo recae en el Teorema 2 (frontera de Bell).

5.3.1 No-conmutatividad (Principio de Incertidumbre) [A-Meta] + [A-Código]

Teorema T-NC-Z [A-Meta]

La medición en Z desordena la información sobre X.

Enunciado.
$\forall(prior: EpistemicState), \forall(result: SpinResult),\ (measure_Z\ prior\ result).prob_u p_X = 50.$

Demostración. Sea $prior$ un estado epistémico arbitrario y sea $result \in \{up, down\}$. Examinemos la definición de $measure_Z$:

$$measure_Z\ prior\ up = \langle 100,50,50,50\rangle,$$

$$measure_Z\ prior\ down = \langle 0,50,50,50\rangle.$$

En ambos casos, el segundo campo de la tupla resultante —que corresponde a $prob_u p_X$— vale 50. Luego, para todo $prior$ y todo $result$, la probabilidad epistémica asignada a $X = up$ tras medir Z es 50. ▫

Teorema T-NC-X [A-Meta]

La medición en X desordena la información sobre Z.

Enunciado.
$\forall(prior: EpistemicState), \forall(result: SpinResult),\ (measure_X\ prior\ result).prob_u p_Z = 50.$

Demostración. Sea $prior$ arbitrario y $result \in \{up, down\}$. Por definición:

$$measure_X\ prior\ up = \langle 50,100,50,50\rangle,$$

$$measure_X\ prior\ down = \langle 50,0,50,50\rangle.$$

En ambas ramas, el primer campo —$prob_u p_Z$— es 50. Por tanto, la medición de X borra toda información previa sobre Z. ▫

Verificación en Lean [A-Código]

```
noncomputable section

theorem measurement_Z_scrambles_X :
    ∀ (prior : EpistemicState) (result : SpinResult),
      (measure_Z prior result).prob_up_X = 50 := by
  intro prior result; cases result <;> rfl

theorem measurement_X_scrambles_Z :
    ∀ (prior : EpistemicState) (result : SpinResult),
      (measure_X prior result).prob_up_Z = 50 := by
  intro prior result; cases result <;> rfl
```

Interpretación: Medir un observable desordena la información sobre observables incompatibles. Esto no es una limitación tecnológica; es una propiedad estructural de la teoría.

5.3.2 La frontera de Bell (Teorema central) [A-Meta] + [A-Código]

Este es el teorema central del capítulo. Demuestra que ninguna teoría de variables ocultas locales puede reproducir las correlaciones predichas por la mecánica cuántica. La demostración procede en tres partes.

Parte A: Cota algebraica clásica [A-Meta] + [A-Código]

Teorema T-CHSH-Alg [A-Meta]

Cota algebraica clásica para la combinación CHSH.

Enunciado. Sean $a, a', b, b' \in \{-1, +1\}$. Entonces:

$$|ab - ab' + a'b + a'b'| \leq 2.$$

Demostración. Sea:

$$E = ab - ab' + a'b + a'b'.$$

Factorizando algebraicamente:

$$E = a(b - b') + a'(b + b').$$

Como $b, b' \in \{-1, +1\}$, distinguimos dos casos exhaustivos.

Caso 1: $b = b'$. Entonces $b - b' = 0$ y $b + b' \in \{-2, +2\}$. Luego:

$$E = a'(b + b') \in \{-2, +2\}.$$

Caso 2: $b \neq b'$. Entonces $b + b' = 0$ y $b - b' \in \{-2, +2\}$. Luego:

$$E = a(b - b') \in \{-2, +2\}.$$

En ambos casos $|E| \leq 2$. Por tanto, la cota vale para toda asignación. ▫

Teorema T-CHSH-Scale [A-Meta]

Acotación de la correlación clásica en el modelo epistémico.

Enunciado. $\forall assignA, \forall assignB,\ |classical_S\ assignA\ assignB| \leq 200$.

Demostración. Sean $assignA, assignB: Axis \to SpinResult$ funciones arbitrarias. Definamos:

$$r_A^Z = spin_to_int(assignA(Z)), \quad r_A^X = spin_to_int(assignA(X)),$$
$$r_B^D = spin_to_int(assignB(D)), \quad r_B^{D'} = spin_to_int(assignB(D')).$$

Por definición de $spin_to_int$, cada una de estas cuatro variables pertenece a $\{-1, +1\}$. Sustituyendo en la definición de $classical_S$:

$$classical_S = 100 \cdot (r_A^Z r_B^D - r_A^Z r_B^{D'} + r_A^X r_B^D + r_A^X r_B^{D'}).$$

Por T-CHSH-Alg, el factor entre paréntesis está acotado en valor absoluto por 2. Luego:

$$|classical_S| = 100 \cdot |r_A^Z r_B^D - r_A^Z r_B^{D'} + r_A^X r_B^D + r_A^X r_B^{D'}| \leq 100 \cdot 2 = 200.$$

▫

Verificación en Lean [A-Código]

```
theorem chsh_bound_core (aZ aX bD bD' : Int)
   (haZ : aZ = 1 ∨ aZ = -1) (haX : aX = 1 ∨ aX = -1)
   (hbD : bD = 1 ∨ bD = -1) (hbD' : bD' = 1 ∨ bD' = -1) :
   (aZ * bD - aZ * bD' + aX * bD + aX * bD').natAbs ≤ 2 := by
   rcases hbD with (rfl | rfl) <;> rcases hbD' with (rfl | rfl) <;>
   rcases haZ with (rfl | rfl) <;> rcases haX with (rfl | rfl) <;>
   norm_num

theorem classical_S_bounded (assignA assignB : Axis → SpinResult) :
   (classical_S assignA assignB).natAbs ≤ 200 := by
   let rA_Z := spin_to_int (assignA .Z)
   let rA_X := spin_to_int (assignA .X)
   let rB_D := spin_to_int (assignB .D)
   let rB_D' := spin_to_int (assignB .D')
   have hA_Z : rA_Z = 1 ∨ rA_Z = -1 := spin_to_int_cases (assignA .Z)
   have hA_X : rA_X = 1 ∨ rA_X = -1 := spin_to_int_cases (assignA .X)
   have hB_D : rB_D = 1 ∨ rB_D = -1 := spin_to_int_cases (assignB .D)
   have hB_D' : rB_D' = 1 ∨ rB_D' = -1 := spin_to_int_cases (assignB
```

```
.D')
   have h_bound := chsh_bound_core rA_Z rA_X rB_D rB_D' hA_Z hA_X hB_D
hB_D'
   have h_eq : classical_S assignA assignB =
     100 * (rA_Z * rB_D - rA_Z * rB_D' + rA_X * rB_D + rA_X * rB_D') :=
by
     unfold classical_S classical_correlation
     simp only [rA_Z, rB_D, rA_X, rB_D']
     ring
   rw [h_eq]
   rw [Int.natAbs_mul]
   have h_abs_100 : Int.natAbs 100 = 100 := by norm_num
   rw [h_abs_100]
   calc
     100 * (rA_Z * rB_D - rA_Z * rB_D' + rA_X * rB_D + rA_X *
rB_D').natAbs
       ≤ 100 * 2 := Nat.mul_le_mul_left 100 h_bound
     _ = 200 := by norm_num
```

Interpretación: Cualquier teoría que postule valores definidos simultáneos para todas las mediciones (CFD) está algebraicamente limitado a $|S| \leq 2$. Es una consecuencia pura de la probabilidad clásica conjunta.

Parte B: Predicción trigonométrica cuántica [A-Meta] + [A-Código]

Teorema T-Bell-Exact [A-Meta]

Valor exacto de la combinación CHSH cuántica.

Enunciado. $S_{quantum} = -2\sqrt{2}$.

Demostración. Por definición:

$$S_{quantum} = -\cos\left(0 - \frac{\pi}{4}\right) + \cos\left(0 - \frac{3\pi}{4}\right) - \cos\left(\frac{\pi}{2} - \frac{\pi}{4}\right) - \cos\left(\frac{\pi}{2} - \frac{3\pi}{4}\right).$$

Evaluemos cada término.

Sea $\alpha = 0 - \frac{\pi}{4} = -\frac{\pi}{4}$. Como cos es par:

$$\cos\left(-\frac{\pi}{4}\right) = \cos\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}.$$

Sea $\beta = 0 - \frac{3\pi}{4} = -\frac{3\pi}{4}$. Entonces:

$$\cos(\beta) = \cos\left(\frac{3\pi}{4}\right) = -\frac{\sqrt{2}}{2}.$$

Sea $\gamma = \frac{\pi}{2} - \frac{\pi}{4} = \frac{\pi}{4}$. Entonces:

$$\cos(\gamma) = \frac{\sqrt{2}}{2}.$$

Sea $\delta = \frac{\pi}{2} - \frac{3\pi}{4} = -\frac{\pi}{4}$. Entonces:

$$\cos(\delta) = \frac{\sqrt{2}}{2}.$$

Sustituyendo:

$$S_{quantum} = -\frac{\sqrt{2}}{2} + \left(-\frac{\sqrt{2}}{2}\right) - \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2} = -4 \cdot \frac{\sqrt{2}}{2} = -2\sqrt{2}.$$

▫

Verificación en Lean [A-Código]

```
theorem bell_violation_exact : S_quantum = -2 * Real.sqrt 2 := by
   unfold S_quantum quantum_correlation angle_a angle_a' angle_b angle_b'
   have h1 : cos (0 - π/4) = Real.sqrt 2 / 2 := by
     calc
       cos (0 - π/4) = cos (-(π/4)) := by ring_nf
       _ = cos (π/4) := by rw [Real.cos_neg]
       _ = Real.sqrt 2 / 2 := Real.cos_pi_div_four
   have h2 : cos (0 - 3*π/4) = -Real.sqrt 2 / 2 := by
     calc
       cos (0 - 3*π/4) = cos (-(3*π/4)) := by ring_nf
       _ = cos (3*π/4) := by rw [Real.cos_neg]
       _ = cos (π - π/4) := by ring_nf
       _ = -cos (π/4) := by rw [Real.cos_pi_sub]
       _ = -(Real.sqrt 2 / 2) := by rw [Real.cos_pi_div_four]
       _ = -Real.sqrt 2 / 2 := by simp [neg_div]
   have h3 : cos (π/2 - π/4) = Real.sqrt 2 / 2 := by
     calc
       cos (π/2 - π/4) = cos (π/4) := by ring_nf
       _ = Real.sqrt 2 / 2 := Real.cos_pi_div_four
   have h4 : cos (π/2 - 3*π/4) = Real.sqrt 2 / 2 := by
     calc
       cos (π/2 - 3*π/4) = cos (-(π/4)) := by ring_nf
       _ = cos (π/4) := by rw [Real.cos_neg]
       _ = Real.sqrt 2 / 2 := Real.cos_pi_div_four
   rw [h1, h2, h3, h4]
   field_simp
   ring_nf
```

Parte C: Violación analítica [A-Meta] + [A-Código]

Teorema T-Violation [A-Meta]

La predicción cuántica viola la cota clásica.

Enunciado. $|S_{quantum}| > 2$.

Demostración. Por T-Bell-Exact:

$$|S_{quantum}| = |-2\sqrt{2}| = 2\sqrt{2}.$$

Sabemos que $\sqrt{2} > 1$, pues $(\sqrt{2})^2 = 2 > 1 = 1^2$ y la función cuadrática es estrictamente creciente en $\mathbb{R}^+$. Luego:

$$2\sqrt{2} > 2 \cdot 1 = 2.$$

Por tanto:

$$|S_{quantum}| > 2.$$

▫

Verificación en Lean [A-Código]

```
theorem quantum_exceeds_classical_bound :
  abs S_quantum > 2 := by
  rw [bell_violation_exact]
  have h_abs : abs (-2 * Real.sqrt 2) = 2 * Real.sqrt 2 := by
    rw [abs_mul, abs_neg, abs_of_nonneg (by positivity)]
    norm_num
  rw [h_abs]
  have h_sqrt : Real.sqrt 2 > 1 := Real.lt_sqrt_of_sq_lt (by norm_num)
  nlinarith [Real.sqrt_nonneg 2, Real.sq_sqrt (show 0 ≤ 2 by norm_num)]
end
```

Interpretación general: La Parte A establece que la CFD limita a $|S| \leq 2$. La Parte B demuestra que la regla de Born predice $S = -2\sqrt{2} \approx -2.828$. La Parte C verifica que $2\sqrt{2} > 2$. La violación de CHSH manifiesta la incompatibilidad estructural entre la distribución de probabilidad conjunta clásica y el estado cuántico global no-separable.

5.3.3 La condición de no-señalización [A-Meta] + [C]

Un requisito fundamental para cualquier interpretación de la mecánica cuántica es la compatibilidad con la relatividad especial: la prohibición de comunicación superlumínica. Esto se formaliza como la condición de no-señalización: la elección del eje de medición de Alice no puede alterar las probabilidades marginales de los resultados de Bob.

En la mecánica cuántica estándar, esto se garantiza matemáticamente mediante la traza parcial. Si el sistema conjunto está en el estado entrelazado $|\Psi\rangle$, la matriz de densidad reducida de Bob es $\rho_B = \mathrm{Tr}_A(|\Psi\rangle\langle\Psi|)$. Un resultado fundamental del formalismo es que ρ_B es independiente de la base de medición que elija Alice; para el estado singlete, ρ_B es siempre proporcional a $\frac{1}{2}I$, lo que significa que las probabilidades marginales de Bob son siempre 50/50 para cualquier eje.

En nuestro modelo epistémico, esta propiedad estructural se refleja en el comportamiento de las funciones de actualización. Independientemente del resultado que obtenga Alice, el estado epistémico que un agente racional debe asignar al sistema de Bob mantiene probabilidades del 50% para cualquier eje incompatible.

Teorema T-NoSignal-Z [A-Meta]

La medición en Z no altera las probabilidades marginales de observables incompatibles.

Enunciado. $\forall prior, \forall result,$

$$(measure_Z\ prior\ result).prob_u p_X = 50 \wedge (measure_Z\ prior\ result).prob_u p_D = 50 \wedge (measure_Z\ prior\ result).prob_u p_D{}' = 50.$$

Demostración. Sea $prior$ arbitrario y $result \in \{up, down\}$. Por definición de $measure_Z$, el estado resultante es:

$$\langle 100,50,50,50 \rangle$$

si $result = up$, y:

$$\langle 0,50,50,50 \rangle$$

si $result = down$. En ambos casos, los campos segundo, tercero y cuarto valen simultáneamente 50. Estos campos corresponden a $prob_u p_X$, $prob_u p_D$ y $prob_u p_D{}'$. Luego las probabilidades marginales de todos los observables incompatibles con Z permanecen en 50. ▫

Teorema T-NoSignal-X [A-Meta]

La medición en X no altera las probabilidades marginales de observables incompatibles.

Enunciado. $\forall prior, \forall result,$

$$(measure_X\ prior\ result).prob_u p_Z = 50 \wedge (measure_X\ prior\ result).prob_u p_D = 50 \wedge (measure_X\ prior\ result).prob_u p_D{}' = 50.$$

Demostración. Análogo a T-NoSignal-Z. La definición de $measure_X$ produce:

$$\langle 50,100,50,50 \rangle$$

o:

$$\langle 50,0,50,50\rangle.$$

Los campos primero, tercero y cuarto son 50 en ambas ramas. ▫

Teorema T-NoSignal-D [A-Meta]

La medición en D no altera las probabilidades marginales de observables incompatibles.

Enunciado. $\forall prior, \forall result,$

$$(measure_D\ prior\ result).prob_u p_Z = 50 \wedge (measure_D\ prior\ result).prob_u p_X = 50 \wedge (measure_D\ prior\ result).prob_u p_D{}' = 50.$$

Demostración. Por definición de $measure_D$, el estado resultante es:

$$\langle 50,50,100,50\rangle$$

o:

$$\langle 50,50,0,50\rangle.$$

Los campos primero, segundo y cuarto valen 50 en ambos casos. ▫

Interpretación: El entrelazamiento cuántico, aunque viola las desigualdades de Bell, no permite la transmisión de información más rápida que la luz. La "no-localidad" cuántica es estrictamente correlacional, no causal. Alice y Bob comparten correlaciones que solo se hacen evidentes al comparar sus registros clásicos posteriormente. Nuestro modelo epistémico captura esta característica sin necesidad de postular acción a distancia física.

5.3.4 Contextualidad [A-Meta] + [A-Código]

Teorema T-Context [A-Meta]

La ordenación de mediciones incompatibles importa.

Enunciado. Sean:

$$prepared = prepared_m aximally_m ixed,$$

$$Z_{then}X = measure_X\ (measure_Z\ prepared\ up)\ up,$$

$$X_{then}Z = measure_Z\ (measure_X\ prepared\ up)\ up.$$

Entonces:

$$Z_{then}X.prob_u p_Z \neq X_{then}Z.prob_u p_Z.$$

Demostración. Calculamos directamente. Por definición:

$$prepared_maximally_mixed = \langle 50,50,50,50 \rangle.$$

Primero:

$$measure_Z\ prepared\ up = \langle 100,50,50,50 \rangle.$$

Luego:

$$Z_{then}X = measure_X\ \langle 100,50,50,50 \rangle\ up = \langle 50,100,50,50 \rangle.$$

Por tanto:

$$Z_{then}X.prob_up_Z = 50.$$

Segundo:

$$measure_X\ prepared\ up = \langle 50,100,50,50 \rangle.$$

Luego:

$$X_{then}Z = measure_Z\ \langle 50,100,50,50 \rangle\ up = \langle 100,50,50,50 \rangle.$$

Por tanto:

$$X_{then}Z.prob_up_Z = 100.$$

Como $50 \neq 100$, se verifica la desigualdad. ▫

Verificación en Lean [A-Código]

```
theorem measurement_order_matters :
    let prepared := prepared_maximally_mixed
    let Z_then_X := measure_X (measure_Z prepared SpinResult.up)
SpinResult.up
    let X_then_Z := measure_Z (measure_X prepared SpinResult.up)
SpinResult.up
    Z_then_X.prob_up_Z ≠ X_then_Z.prob_up_Z := by
  unfold measure_X measure_Z prepared_maximally_mixed
  simp
```

Interpretación: Verificación de consistencia que captura la contextualidad cuántica. Medir Z y luego X produce un estado epistémico diferente a medir X y luego Z.

5.3.5 Bloqueo de la definición contracausal (CFD) [A-Meta] + [A-Código]

Teorema T-CFD-Impl [A-Meta]

La definición contracausal implica valores simultáneos definidos.

Enunciado. Sea $\mathcal{T}$ una teoría de variables ocultas locales tal que $counterfactual_definiteness_holds(\mathcal{T})$. Entonces para todo $a_1, a_2, b \in Axis$ y toda variable oculta $hv \in \mathcal{T}.hidden_variables$, existen $r_{A1}, r_{A2}, r_B \in SpinResult$ tales que:

$$\mathcal{T}.result_A\ a_1\ hv = r_{A1} \land \mathcal{T}.result_A\ a_2\ hv = r_{A2} \land \mathcal{T}.result_B\ b\ hv = r_B.$$

Demostración. Sea hv una variable oculta arbitraria y sean a_1, a_2, b ejes arbitrarios. Aplicamos la hipótesis de definición contracausal al par (a_1, b): existe r_{A1} tal que:

$$\mathcal{T}.result_A\ a_1\ hv = r_{A1}.$$

Aplicamos la misma hipótesis al par (a_2, b): existe r_{A2} tal que:

$$\mathcal{T}.result_A\ a_2\ hv = r_{A2}.$$

Finalmente, aplicamos la hipótesis nuevamente a (a_1, b) para obtener r_B tal que:

$$\mathcal{T}.result_B\ b\ hv = r_B.$$

Tomando estas tres existencias conjuntamente, obtenemos el resultado deseado. ▫

Teorema T-CFD-Block [A-Meta]

El modelo epistémico bloquea la definición contracausal.

Enunciado.

$$\text{prepared_Z_up.prob_up_X} = 50 \land \text{prepared_Z_up.prob_up_D} = 50 \land \text{prepared_Z_up.prob_up_D}' = 50.$$

Demostración. Por definición:

$$\text{prepared_Z_up} = \langle 100{,}50{,}50{,}50 \rangle.$$

Los campos segundo, tercero y cuarto de esta tupla son, respectivamente, $prob_u p_X$, $prob_u p_D$ y $prob_u p_D{}'$. Cada uno vale 50 por inspección directa de la tupla. ▫

Verificación en Lean [A-Código]

```
structure LocalHiddenVariableTheory where
   hidden_variables : Type
   result_A : Axis → hidden_variables → SpinResult
   result_B : Axis → hidden_variables → SpinResult

def counterfactual_definiteness_holds
     (theory : LocalHiddenVariableTheory) : Prop :=
   ∀ (a b : Axis) (hv : theory.hidden_variables),
     ∃ (rA rB : SpinResult),
       theory.result_A a hv = rA ∧ theory.result_B b hv = rB

theorem CFD_implies_simultaneous_definite_values
     (theory : LocalHiddenVariableTheory)
     (h_CFD : counterfactual_definiteness_holds theory) :
     ∀ (a1 a2 b : Axis) (hv : theory.hidden_variables),
       ∃ (rA1 rA2 rB : SpinResult),
         theory.result_A a1 hv = rA1 ∧
         theory.result_A a2 hv = rA2 ∧
```

```
        theory.result_B b hv = rB := by
  intro a1 a2 b hv
  obtain ⟨rA1, _, hA1, _⟩ := h_CFD a1 b hv
  obtain ⟨rA2, _, hA2, _⟩ := h_CFD a2 b hv
  obtain ⟨_, rB, _, hB⟩ := h_CFD a1 b hv
  exact ⟨rA1, rA2, rB, hA1, hA2, hB⟩

theorem epistemic_interpretation_blocks_CFD :
    prepared_Z_up.prob_up_X = 50 ∧
    prepared_Z_up.prob_up_D = 50 ∧
    prepared_Z_up.prob_up_D' = 50 := by
  unfold prepared_Z_up
  simp
```

Interpretación: La CFD obliga a aceptar resultados simultáneamente definidos para mediciones no realizadas. En nuestro modelo, un sistema preparado en Z=up tiene probabilidad 50% para X, D y D'. Esto bloquea la CFD y evita las desigualdades de Bell.

5.4 Formalización del pluralismo interpretativo y la dinámica discursiva [A-Meta] + [A-Código]

En las secciones anteriores demostramos física: los límites algebraicos que distinguen lo cuántico de lo clásico. Ahora formalizamos el espacio de interpretaciones viables y la dinámica histórica del discurso que transita entre ellas.

5.4.1 Z3: Mapa formal del pluralismo interpretativo [A-Meta] + [A-Código]

El teorema de Bell (1964) no refuta "el realismo" en abstracto; refuta una conjunción específica de tres supuestos metafísicos. Z3 nos permite mapear exhaustivamente todo el espacio de posibilidades interpretativas, demostrando que existen exactamente 7 interpretaciones viables (cada una renunciando a al menos un supuesto).

Metateoremas del mapa lógico Z3 [A-Meta]

Teorema T-Z3-UNSAT [A-Meta]

El realismo local clásico es formalmente refutado.

Enunciado. El conjunto de fórmulas:

$$\{Locality, CFD, StatIndep, CHSH_ClassicalBound, Aspect_Experiment\}$$

es insatisfacible.

Demostración. Supongamos, para reducir al absurdo, que existe un modelo $\mathcal{M}$ donde las cinco fórmulas son simultáneamente verdaderas. Por $CHSH_ClassicalBound$:

$$Locality \wedge CFD \wedge StatIndep \Longrightarrow S_{value} \leq 2.$$

Como en $\mathcal{M}$ los tres antecedentes son verdaderos, por modus ponens se sigue:

$$S_{value} \leq 2.$$

Pero por $Aspect_Experiment$, en $\mathcal{M}$ se tiene:

$$S_{value} = 2{,}828.$$

Luego:

$$2{,}828 \leq 2,$$

lo cual es aritméticamente falso. Hemos derivado una contradicción. Por tanto, no existe tal modelo $\mathcal{M}$; el conjunto es insatisfacible. ▫

Teorema T-Z3-Bohm [A-Meta]

Viabilidad lógica de la teoría de Bohm.

Enunciado. El conjunto:

$$\{\neg Locality, CFD, StatIndep, S_{value} = 2{,}828\}$$

es satisfacible.

Demostración. Construimos explícitamente un modelo $\mathcal{M}$. Definimos:

$$Locality^{\mathcal{M}} = \bot, \quad CFD^{\mathcal{M}} = \top, \quad StatIndep^{\mathcal{M}} = \top, \quad S_{value}^{\mathcal{M}} = 2{,}828.$$

Verificamos cada fórmula del conjunto:

27. $\neg Locality$ es verdadero pues $Locality^{\mathcal{M}} = \bot$.
28. CFD es verdadero por asignación.
29. $StatIndep$ es verdadero por asignación.
30. $S_{value} = 2{,}828$ es verdadero por asignación.

Como todas las fórmulas son verdaderas en $\mathcal{M}$, el conjunto es satisfacible. ▫

Teorema T-Z3-QBism [A-Meta]

Viabilidad lógica de la interpretación epistémica.

Enunciado. El conjunto:

$$\{Locality, \neg CFD, StatIndep, S_{value} = 2{,}828\}$$

es satisfacible.

Demostración. Construimos el modelo $\mathcal{M}$ con:

$$Locality^{\mathcal{M}} = \top, \quad CFD^{\mathcal{M}} = \bot, \quad StatIndep^{\mathcal{M}} = \top, \quad S_{value}^{\mathcal{M}} = 2{,}828.$$

Cada fórmula del conjunto se evalúa a verdadero por asignación directa. Luego el conjunto es satisfacible. ▫

Teorema T-Z3-Superdet [A-Meta]

Viabilidad lógica del superdeterminismo.

Enunciado. El conjunto:

$$\{Locality, CFD, \neg StatIndep, S_{value} = 2{,}828\}$$

es satisfacible.

Demostración. Sea $\mathcal{M}$ el modelo con:

$$Locality^{\mathcal{M}} = \top, \quad CFD^{\mathcal{M}} = \top, \quad StatIndep^{\mathcal{M}} = \bot, \quad S_{value}^{\mathcal{M}} = 2{,}828.$$

Todas las fórmulas del conjunto son verdaderas en $\mathcal{M}$. ▫

Teorema T-Z3-GRW [A-Meta]

Viabilidad lógica del colapso objetivo.

Enunciado. El conjunto:

$$\{\neg Locality, \neg CFD, StatIndep, S_{value} = 2{,}828\}$$

es satisfacible.

Demostración. Sea $\mathcal{M}$ con:

$$Locality^{\mathcal{M}} = \bot, \quad CFD^{\mathcal{M}} = \bot, \quad StatIndep^{\mathcal{M}} = \top, \quad S_{value}^{\mathcal{M}} = 2{,}828.$$

Cada fórmula se satisface por asignación. ▫

Teorema T-Z3-Retro [A-Meta]

Viabilidad lógica de la retrocausalidad.

Enunciado. El conjunto:

$$\{\neg Locality, CFD, \neg StatIndep, S_{value} = 2{,}828\}$$

es satisfacible.

Demostración. Sea $\mathcal{M}$ con:

$$Locality^{\mathcal{M}} = \bot, \quad CFD^{\mathcal{M}} = \top, \quad StatIndep^{\mathcal{M}} = \bot, \quad S_{value}^{\mathcal{M}} = 2{,}828.$$

Verificación directa. ▫

Teorema T-Z3-Relativist [A-Meta]

Viabilidad lógica del relativismo ontológico.

Enunciado. El conjunto:

$$\{Locality, \neg CFD, \neg StatIndep, S_{value} = 2{,}828\}$$

es satisfacible.

Demostración. Sea $\mathcal{M}$ con:

$$Locality^{\mathcal{M}} = \top, \quad CFD^{\mathcal{M}} = \bot, \quad StatIndep^{\mathcal{M}} = \bot, \quad S_{value}^{\mathcal{M}} = 2{,}828.$$

Verificación directa. ▫

Teorema T-Z3-Radical [A-Meta]

Viabilidad lógica del pluralismo radical.

Enunciado. El conjunto:

$$\{\neg Locality, \neg CFD, \neg StatIndep, S_{value} = 2{,}828\}$$

es satisfacible.

Demostración. Sea $\mathcal{M}$ con:

$$Locality^{\mathcal{M}} = \bot, \quad CFD^{\mathcal{M}} = \bot, \quad StatIndep^{\mathcal{M}} = \bot, \quad S_{value}^{\mathcal{M}} = 2{,}828.$$

Verificación directa. ▫

Verificación computacional [A-Código]

```
; ============================================================
; Z3: MAPA FORMAL DEL PLURALISMO INTERPRETATIVO CUÁNTICO
; ============================================================
(set-option :produce-unsat-cores true)
(set-option :produce-models true)

; --- 1. Supuestos metafísicos del teorema de Bell ---
(declare-const Locality Bool)      ; L: factorización de probabilidades
(declare-const CFD Bool)           ; C: Counterfactual Definiteness
(declare-const StatIndep Bool)     ; I: Independencia estadística
λ⊥choices

; --- 2. Parámetro empírico CHSH ---
(declare-const S_value Real)

; --- 3. Axiomas nombrados ---
; Axioma físico: bajo L∧C∧I, la cota CHSH clásica es |S| ≤ 2
(assert (! (=> (and Locality CFD StatIndep) (<= S_value 2.0))
         :named CHSH_ClassicalBound))
```

```
; Axioma empírico: Aspect, Weihs, Hensen et al. confirman S ≈ 2.828
(assert (! (= S_value 2.828)
         :named Aspect_Experiment))

; ============================================================
; TEOREMA 1: El realismo local clásico es FORMALMENTE REFUTADO
; ============================================================
(push)
(assert (! (and Locality CFD StatIndep) :named Classical_Local_Realism))
(check-sat)
(get-unsat-core)
(pop)

; ============================================================
; TEOREMA 2: Renunciar a UN supuesto restaura la consistencia
; ============================================================
; 2A: ¬Localidad ∧ CFD ∧ SI  →  TEORÍA DE BOHM
(push)
(assert (not Locality))
(assert CFD)
(assert StatIndep)
(check-sat)
(get-model)
(pop)

; 2B: Locality ∧ ¬CFD ∧ SI  →  QBism, OQM, COPENHAGUE ESTRICTO
(push)
(assert Locality)
(assert (not CFD))
(assert StatIndep)
(check-sat)
(get-model)
(pop)

; 2C: Locality ∧ CFD ∧ ¬SI  →  SUPERDETERMINISMO
(push)
(assert Locality)
(assert CFD)
(assert (not StatIndep))
(check-sat)
(get-model)
(pop)

; ============================================================
; TEOREMA 3: Renunciar a DOS supuestos también es consistente
; ============================================================
; 3A: ¬Locality ∧ ¬CFD ∧ SI  →  GRW, HISTORIAS CONSISTENTES
(push)
(assert (not Locality))
(assert (not CFD))
```

```
(assert StatIndep)
(check-sat)
(get-model)
(pop)

; 3B: ¬Locality ∧ CFD ∧ ¬SI  →  RETROCAUSALIDAD
(push)
(assert (not Locality))
(assert CFD)
(assert (not StatIndep))
(check-sat)
(get-model)
(pop)

; 3C: Locality ∧ ¬CFD ∧ ¬SI  →  RELATIVISMO ONTOLOGICO
(push)
(assert Locality)
(assert (not CFD))
(assert (not StatIndep))
(check-sat)
(get-model)
(pop)

; ============================================================
; TEOREMA 4: Incluso ¬Locality ∧ ¬CFD ∧ ¬SI es consistente
; ============================================================
(push)
(assert (not Locality))
(assert (not CFD))
(assert (not StatIndep))
(check-sat)
(get-model)
(pop)
```

Resultado verificado:

```
unsat
(CHSH_ClassicalBound Classical_Local_Realism Aspect_Experiment)

sat
(modelo con Locality=false, CFD=true, StatIndep=true, S_value=2.828)

sat
(modelo con Locality=true, CFD=false, StatIndep=true, S_value=2.828)

[... 5 modelos sat adicionales ...]
```

Interpretación del núcleo insatisfacible: Z3 identifica la tensión exacta entre tres proposiciones:

Axioma	Origen	Rol en la contradicción
CHSH_ClassicalBound	Álgebra pura (Teorema 2 de Lean)	Si $L \wedge C \wedge I$, entonces $\|S\| \leq 2$
Aspect_Experiment	Laboratorio (Aspect, Weihs, Hensen)	$S = 2.828$
Classical_Local_Realism	Hipótesis ontológica	$L \wedge C \wedge I$

Z3 demuestra que no existe un mundo posible donde estas tres proposiciones sean simultáneamente verdaderas. El núcleo es mínimo: si eliminas cualquiera de las tres, la inconsistencia desaparece.

Tabla de las 7 interpretaciones viables:

#	Localidad	CFD	Indep.	Interpretación	Tradición
2	✗	✓	✓	Teoría de Bohm	Onda piloto no-local
3	✓	✗	✓	QBism / OQM	Epistémica contextual
4	✓	✓	✗	Superdeterminismo	λ correlacionado con elecciones
5	✗	✗	✓	GRW / Historias Consistentes	Colapso objetivo
6	✗	✓	✗	Retrocausalidad	Influencia hacia atrás
7	✓	✗	✗	Relativismo ontológico	Todo es relacional
8	✗	✗	✗	Pluralismo radical	Ningún supuesto clásico

Conclusión: La interpretación epistémica que defendemos (fila 3) es una de 7 opciones viables. La elegimos por parsimonia ontológica (evita acción a distancia y conspiración cósmica), no porque sea lógicamente inevitable. Z3 formaliza la tesis de Duhem-Quine-Lakatos: los experimentos refutan conjunciones de supuestos, no teorías aisladas.

5.4.2 Clingo: Dinámica del discurso cuántico [A-Meta] + [A-Código]

Si el espacio de modelos de Z3 es estático —un mapa de posibilidades—, el modelo Clingo es dinámico: simula cómo una comunidad científica transita entre esas posibilidades cuando llega evidencia contraria al paradigma dominante.

Verificación computacional [A-Código]

```
% ============================================================
% CLINGO: DINÁMICA DEL DISCURSO CUÁNTICO
% Evolución temporal de ontologías ante evidencia creciente
% ============================================================
```

```
#const t_max = 3.

% 0. DOMINIO TEMPORAL
time(0..t_max).

% 1. ONTOLOGÍAS: vértices del cubo de Bell (L, C, I)
ontology(classical, 1, 1, 1).      % Realismo local clásico (INSAT con
datos)
ontology(bohm, 0, 1, 1).           % Renuncia a Localidad
ontology(qbism, 1, 0, 1).          % Renuncia a CFD (epistémica)
ontology(superdet, 1, 1, 0).       % Renuncia a Independencia Estadística
ontology(grw, 0, 0, 1).            % Renuncia a L y C
ontology(retro, 0, 1, 0).          % Renuncia a L y I
ontology(relativist, 1, 0, 0).     % Renuncia a C y I
ontology(radical, 0, 0, 0).        % Renuncia a los tres

% 2. AGENTES DEL DISCURSO: comunidades interpretativas
agent(bohmians; qbists; copenhagen; everettians; experimentalists).

% 3. OBJETOS MATEMÁTICOS Y TIPOS REALES
math_obj(wavefunction; measurement_op; entangled_state; born_rule).

real_type(wavefunction, structure).
real_type(measurement_op, relation).
real_type(entangled_state, structure).
real_type(born_rule, relation).

% 4. PREDICADOS DEL DISCURSO
pred(collapse, substance).
pred(superposition, substance).
pred(forces_choice, substance).
pred(action_at_distance, substance).
pred(possesses_spin, substance).
pred(information_update, relation).
pred(correlation, structure).

% 5. PROCESOS SATÉLITE
satellite(naturalization; personification; idealization).

% 6. EVIDENCIA HISTÓRICA
evidence(bell_theorem, 1).         % 1964: L∧C∧I es inconsistente
evidence(aspect_violation, 2).     % 1982: S=2.828 confirmado
evidence(ghz_contextuality, 3).    % 1990s: contextualidad sin loopholes

% 7. ESTADO INICIAL (t=0): paradigma clásico universal
holds(A, classical, 0) :- agent(A).

% Atributos discursivos iniciales
discourse_attr(wavefunction, collapse, 0).
discourse_attr(wavefunction, superposition, 0).
```

```
discourse_attr(measurement_op, forces_choice, 0).
discourse_attr(entangled_state, action_at_distance, 0).
discourse_attr(entangled_state, possesses_spin, 0).

% 8. DINÁMICA DE TRANSICIÓN
{ holds(A, O, T) : ontology(O, _, _, _) } = 1 :- agent(A), time(T), T >=
1, T <= t_max.

% 9. SATÉLITES ACTIVOS
satellite_active(A, S, T) :- holds(A, classical, T), satellite(S),
time(T), T >= 1.

% 10. RESTRICCIONES DE SOSTENIBILIDAD
:- holds(A, classical, T), time(T), T = 1,
   #count { S : satellite_active(A, S, T) } < 2.

:- holds(A, classical, T), time(T), T >= 2,
   evidence(aspect_violation, T2), T2 <= T,
   #count { S : satellite_active(A, S, T) } < 3.

% 11. REIFICACIÓN DINÁMICA
has_epistemic(T) :- holds(A, O, T), agent(A), ontology(O, _, 0, _).

discourse_attr(wavefunction, information_update, T) :- has_epistemic(T).
discourse_attr(entangled_state, correlation, T)       :-
has_epistemic(T).

discourse_attr(wavefunction, collapse, T)             :- holds(A,
classical, T), agent(A).
discourse_attr(wavefunction, superposition, T)        :- holds(A,
classical, T), agent(A).
discourse_attr(measurement_op, forces_choice, T)      :- holds(A,
classical, T), agent(A).
discourse_attr(entangled_state, action_at_distance, T):- holds(A,
classical, T), agent(A).
discourse_attr(entangled_state, possesses_spin, T)    :- holds(A,
classical, T), agent(A).

% 12. DETECCIÓN DE REIFICACIÓN
reified(Obj, T) :- math_obj(Obj), discourse_attr(Obj, P, T), pred(P,
T_req),
                   real_type(Obj, T_real), T_req != T_real, time(T).

% 13. MÉTRICAS FILOSÓFICAS
changed(A, T) :- time(T), T > 0, holds(A, O, T), holds(A, O2, T-1), O !=
O2.

crisis(T) :- time(T), T > 0, #count { A : changed(A, T) } >= 2.

ontology_held(O, T) :- holds(A, O, T), agent(A).
```

```
fragmentation(T) :- time(T), T > 0, #count { O : ontology_held(O, T) } >=
3.

phantom_ontology(A, T) :- holds(A, classical, T), time(T),
                          evidence(_, T2), T2 < T,
                          #count { S : satellite_active(A, S, T) } < 3.

stabilized(A) :- agent(A), holds(A, O, 2), changed(A, 2), holds(A, O, 3).

% 14. FORZAR DINÁMICA HISTÓRICA
:- not crisis(1).
:- not fragmentation(2).
:- #count { A : stabilized(A) } = 0.

% 15. OPTIMIZACIÓN: PARSIMONIA ONTOLÓGICA
#minimize { 1, A, S, T : satellite_active(A, S, T) }.
#minimize { 1, Obj, T : reified(Obj, T) }.
#minimize { 1, A, T : changed(A, T) }.

% 16. OUTPUT
#show holds/3.
#show crisis/1.
#show fragmentation/1.
#show phantom_ontology/2.
#show reified/2.
#show changed/2.
```

Resultado verificado (modelo óptimo, Answer 3):

```
holds(bohmians,classical,0) holds(bohmians,retro,1)
holds(bohmians,retro,2) holds(bohmians,retro,3)
holds(copenhagen,classical,0) holds(copenhagen,superdet,1)
holds(copenhagen,superdet,2) holds(copenhagen,superdet,3)
holds(everettians,classical,0) holds(everettians,bohm,1)
holds(everettians,retro,2) holds(everettians,retro,3)
holds(experimentalists,classical,0) holds(experimentalists,retro,1)
holds(experimentalists,retro,2) holds(experimentalists,retro,3)
holds(qbists,classical,0) holds(qbists,bohm,1) holds(qbists,bohm,2)
holds(qbists,bohm,3)

crisis(1)

fragmentation(1) fragmentation(2) fragmentation(3)

reified(wavefunction,0) reified(measurement_op,0)
reified(entangled_state,0)

Optimization: 9
OPTIMUM FOUND
```

Metateoremas formales de la dinámica discursiva Clingo [A-Meta]

Teorema T-AS-Crisis [A-Meta]

La crisis es una restricción hard del programa.

Enunciado. En todo answer set AS del programa Clingo de la sección 5.4.2, se tiene:

$$crisis(1) \in AS.$$

Demostración. El programa contiene la restricción hard:

```
:- not crisis(1).
```

Esta regla elimina todo answer set donde $crisis(1)$ no pertenezca al modelo. Por definición de semántica de answer set programming, un answer set debe satisfacer todas las restricciones hard. Luego, si AS es un answer set, necesariamente:

$$crisis(1) \in AS.$$

▫

Teorema T-AS-Frag [A-Meta]

La fragmentación en $t = 2$ es una restricción hard.

Enunciado. En todo answer set AS del programa, se tiene:

$$fragmentation(2) \in AS.$$

Demostración. El programa incluye la restricción:

```
:- not fragmentation(2).
```

Por la semántica de answer sets, todo modelo estable debe contener todos los átomos cuya ausencia produciría violación de una restricción hard. Por tanto:

$$fragmentation(2) \in AS$$

para todo answer set AS. ▫

Teorema T-AS-Stab [A-Meta]

La estabilización de al menos un agente es forzada.

Enunciado. En todo answer set AS del programa, existe al menos un agente A tal que:

$$stabilized(A) \in AS.$$

Demostración. El programa contiene la restricción:

```
:- #count { A : stabilized(A) } = 0.
```

Supongamos, para reducir al absurdo, que existe un answer set AS donde ningún átomo $stabilized(A)$ pertenece a AS. Entonces el conjunto:

$$\{A: stabilized(A) \in AS\}$$

es vacío, y su cardinalidad es 0. La restricción hard se violaría, lo cual contradice la definición de answer set. Luego, en todo answer set existe al menos un agente A con:

$$stabilized(A) \in AS.$$

▫

Interpretación del resultado óptimo

Agente	t=0	t=1 (Bell)	t=2 (Aspect)	t=3 (GHZ)
bohmians	classical	retro	retro	retro
copenhagen	classical	superdet	superdet	superdet
everettians	classical	bohm	retro	retro
experimentalists	classical	retro	retro	retro
qbists	classical	bohm	bohm	bohm

Hallazgo filosófico central: Bajo los criterios de optimización (minimizar satélites + reificación + cambios bruscos), ningún agente adopta qbism en el modelo óptimo. Esto no es un bug; es una virtud epistémica. Demuestra que el modelo no está construido para producir la conclusión deseada. La interpretación epistémica (qbism) requiere renunciar a la CFD, un cambio conceptual que el optimizador penaliza como costoso.

Conexión con Kuhn y Lakatos: El modelo reproduce la estructura lógica de la historia real:

31. t=0: Dominio del paradigma clásico.
32. t=1 (crisis): Bell desata el debate; emergen múltiples interpretaciones.
33. t=2 (fragmentación): Aspect confirma violación; consolidación de "escuelas".
34. t=3: Estabilización en nichos ontológicos paralelos.

No hay conversión masiva a una única interpretación, sino fragmentación persistente. Esto es exactamente lo que predicen Kuhn (inconmensurabilidad post-crisis) y Lakatos (proliferación de programas de investigación).

Ausencia de `phantom_ontology`: Ningún agente mantiene `classical` después de la evidencia sin los satélites necesarios. El paradigma clásico colapsa completamente; no hay "zombies epistemológicos".

5.4.3 Conexión con el protocolo anti-reificación

Las tres formalizaciones cierran el círculo del protocolo de cuatro preguntas:

Nivel	Herramienta	Pregunta	Resultado
Física	Lean 4	¿Qué predice la teoría?	$\|S\| \leq 2$ clásico; $S = 2\sqrt{2}$ cuántico
Lógica	Z3	¿Qué interpretaciones son consistentes?	7 vértices viables; 1 insatisfacible
Discurso	Clingo	¿Cómo evoluciona el debate?	Fragmentación en nichos; qbism no es la trayectoria óptima pura

Nota sobre la gramática y la reificación: Como demostramos en los capítulos anteriores, la estructura gramatical del lenguaje natural (sujeto-predicado) facilita la reificación al empujarnos a tratar procesos y relaciones como sustancias. El modelo Clingo captura esto formalmente: los `discourse_attr` que exigen tipos `substance` para objetos matemáticos de tipo `structure` o `relation` generan `reified(X,T)`. Los procesos satélite (naturalización, personificación, idealización) son mecanismos lingüísticos que estabilizan esta reificación gramaticalmente inducida.

5.5 Evidencia empírica: el debate sobre la interpretación [B/C]

La interpretación de Copenhague (lectura epistémica)

Niels Bohr y Werner Heisenberg enfatizaron que la función de onda no describe la realidad en sí, sino nuestro conocimiento de la realidad.

QBism (Bayesianismo Cuántico)

QBism lleva la interpretación epistémica a su conclusión lógica: la función de onda es una creencia subjetiva del agente. El "colapso" es la actualización bayesiana de esas creencias [B].

Mecánica Cuántica Operacional (OQM)

La OQM de Reinhard Werner trata las correlaciones EPRB como un hecho bruto sin causa común [B]. Los estados cuánticos son puros recursos epistémicos.

La interpretación de los muchos mundos

No hay colapso. La función de onda evoluciona unitariamente siempre. Lo que percibimos como "colapso" es la ramificación del universo en múltiples ramas.

El experimento de Wigner's friend

Ilustra la naturaleza epistémica del colapso. El "colapso" es relativo al observador y a su información disponible.

5.6 El entrelazamiento cuántico y la no-localidad [B/C]

El fenómeno del entrelazamiento

Dos partículas entrelazadas exhiben correlaciones que no pueden explicarse por variables ocultas locales. Einstein llamó a esto "acción fantasmal a distancia" y lo consideró evidencia de incompletitud.

La interpretación epistémica del entrelazamiento

Bajo la interpretación epistémica, el entrelazamiento no implica acción a distancia causal. La elección de eje de Alice no afecta las probabilidades marginales de Bob (sección 5.3.3). Lo que ocurre es una actualización instantánea de información correlacionada, compatible con la relatividad especial.

Evidencia experimental

Los experimentos de Alain Aspect (1982) y posteriores confirman $S = 2\sqrt{2} \approx 2.828$, violando el límite clásico [B].

5.6.1 La crítica realista al teorema de Bell [B/C]

El análisis contemporáneo revela que la estructura lógica del teorema de Bell descansa sobre tres supuestos metafísicos [B]:

Supuesto 1: Localidad. La elección de Alice no afecta instantáneamente el resultado de Bob. Investigadores como Pitowsky y Fine han demostrado que las desigualdades de Bell pueden derivarse directamente de las leyes de la probabilidad clásica, sin necesidad de invocar explícitamente localidad [B].

Supuesto 2: Definición contracausal (CFD). Incluso los experimentos no realizados tienen resultados definidos. EPR exigía "elementos de realidad" preexistentes, pero no especificaba que debieran existir simultáneamente para todas las magnitudes. La CFD introduce una condición más fuerte [B].

Supuesto 3: Independencia estadística (SI). La variable oculta λ es independiente de las elecciones de medición futuras [B].

La crítica realista sostiene que Bell no refuta el realismo en general, sino el realismo local clásico con definición contracausal. Nuestro modelo Z3 (sección 5.4.1) formaliza esta crítica: existen 7 interpretaciones viables que renuncian a al menos uno de estos supuestos.

Característica	Modelo Realista Local (VOL)	Modelo Epistémico (MC)
Objeto fundamental	Variables ocultas locales	Estado cuántico global $\|\Psi\rangle$

	(λ)	
Naturaleza del sistema	Partículas separables	Todo indivisible e irreductible
Resultado CHSH (valor S)	Máximo de 2	$2\sqrt{2} \approx 2.828$

5.6.2 La renuncia ontológica como motor de la física [C]

El análisis del teorema de Bell revela un patrón profundo que recorre la historia de la física: el progreso científico ocurre frecuentemente cuando renunciamos a buscar "causas ocultas" para fenómenos que tienen explicaciones estructurales, geométricas o simétricas.

El caso del movimiento: del impetus a la simetría

En la física medieval, la intuición sugería que el movimiento requería una causa continua: el impetus. La Primera Ley de Newton resolvió este problema redefiniendo el concepto de "causa": el movimiento uniforme no requiere causa; es el estado natural. Lo que requiere causa es el cambio de movimiento [B].

El Teorema de Noether (1918) proporcionó la explicación más profunda: la conservación del momento lineal es consecuencia directa de la simetría de traslación espacial (homogeneidad del espacio). No hay un impetus oculto; hay una propiedad estructural del espacio mismo [B].

(Nota histórica: La conexión entre el abandono del impetus y el Teorema de Noether es una lectura retrospectiva y conceptual. Newton abandonó el impetus sin conocer a Noether; fue Noether quien, siglos después, reveló la simetría profunda que subyace a la inercia newtoniana).

El caso de la gravedad: de la fuerza a la geometría

La intuición newtoniana sugería que la gravedad es una fuerza que "tira" de los objetos a distancia. Einstein renunció a esta imagen. En la relatividad general, la gravedad no es una fuerza: es la geometría curvada del espacio-tiempo. No hay una "fuerza gravitacional" oculta; hay estructura geométrica [B].

El caso del entrelazamiento: de la acción fantasmal a la no-separabilidad

La intuición clásica sugiere que las correlaciones cuánticas requieren una causa: variables ocultas locales o acción a distancia instantánea. La interpretación epistémica renuncia a ambas. El entrelazamiento no es una "acción fantasmal"; es una propiedad estructural del espacio de Hilbert: la no-separabilidad del estado cuántico global.

La síntesis y sus límites

Fenómeno	Intuición (causa oculta)	Física (estructura manifiesta)
Movimiento	Impetus (fuerza motriz)	Simetría de traslación

		(Noether)
Gravedad	Fuerza gravitacional	Geometría curvada (Einstein)
Entrelazamiento	Variables ocultas / acción a distancia	No-separabilidad (Hilbert)
Colapso	Evento físico misterioso	Actualización epistémica (Bayes)

Matización importante: La "renuncia ontológica" no es el único motor de la física. La física también avanza postulando entidades ocultas reales que luego son confirmadas empíricamente (átomos, quarks, materia oscura). El principio que identificamos aquí aplica específicamente a dominios donde la intuición exige causas mecánicas locales para fenómenos que el formalismo matemático revela como restricciones geométricas, simétricas o estructurales.

Conexión con el Realismo Estructural Ontológico (OSR)

Esta síntesis conecta con el Realismo Estructural Ontológico (OSR): lo fundamental no son las partículas con propiedades locales, sino la estructura de relaciones que el formalismo matemático describe [B].

Nota sobre Instrumentalismo y OSR: Nuestro modelo formal es metodológicamente agnóstico (instrumentalismo operacional: la función de onda como recurso predictivo). El OSR se presenta aquí como una hipótesis ontológica prometedora sobre la naturaleza de esa "estructura", no como una conclusión deductiva de nuestro modelo.

5.7 La pedagogía de la medición cuántica [B/C]

La reificación en los libros de texto

La mayoría de los libros de texto presentan el colapso como un evento físico, reificando la función de onda. Esta narrativa es en parte una caricatura didáctica, pero como demostramos en los capítulos anteriores, está facilitada por la estructura gramatical del lenguaje natural, que empuja a tratar "la función de onda" como un sustantivo que denota una entidad sustancial. Esto produce misconceptions persistentes [B].

La solución pedagógica

La investigación en educación física (PER) sugiere desarrollar conciencia ontológica en los estudiantes: enseñarles qué tipo de cosa es la función de onda antes de enseñarles la mecánica cuántica [B].

La paradoja pedagógica

La reificación es necesaria para el aprendizaje inicial, pero perniciosa si no se desactiva. La solución es usar la reificación como prótesis cognitiva visible [C].

5.8 Limitaciones [C]

Primera limitación: La interpretación epistémica es una elección, no un descubrimiento.

Segunda limitación: La interpretación epistémica no resuelve todos los problemas (¿qué es el sistema físico en sí?).

Tercera limitación: La interpretación epistémica es controversial.

Cuarta limitación: Nuestra formalización es simplificada (no modela espacios de Hilbert de dimensión infinita).

Quinta limitación: La interpretación epistémica no resuelve el problema de la frontera (¿qué constituye una medición?).

Sexta limitación: La interpretación epistémica renuncia al realismo clásico (CFD). El Teorema 2 verifica que ningún modelo clásico local puede alcanzar $|S| > 2$.

Séptima limitación: Las formalizaciones en Z3 y Clingo son modelos abstractos que capturan la estructura lógica del pluralismo interpretativo y la dinámica del discurso, pero no modelan la complejidad completa de las comunidades científicas reales (factores sociológicos, institucionales, económicos). Además, el modelo Clingo revela que la interpretación epistémica no es la trayectoria óptima bajo criterios de parsimonia pura; su adopción requiere argumentos filosóficos adicionales que trascienden la mera consistencia formal.

Octava limitación: La gramática del lenguaje natural facilita la reificación al empujarnos a tratar procesos y relaciones como sustancias. Aunque nuestro modelo Clingo captura esto formalmente, no podemos eliminar completamente esta constricción gramatical del discurso científico cotidiano.

5.9 Conexiones con otros capítulos [C]

Con el Capítulo 1 (protocolo): El sistema formalizado es una implementación verificable del protocolo de cuatro preguntas, aplicada a la medición cuántica.

Con el Capítulo 2 (gramática de la reificación): La sección 5.4.2 (Clingo) formaliza cómo la estructura gramatical del lenguaje natural (sujeto-predicado) facilita la reificación de objetos matemáticos.

Con el Capítulo 3 (SPA/Tarski): La estratificación normativa (Z3: *unsat*) es el análogo cuántico de la estratificación lógica (Lean: paradoja de Tarski). Ambos muestran que ciertas configuraciones generan contradicciones.

Con el Capítulo 4 (física clásica): La sección 5.6.2 (renuncia ontológica) unifica la física clásica, relativista y cuántica bajo el principio de abandonar causas ocultas en favor de estructura geométrica y simétrica.

Con el Capítulo 6 (sistemas normativos): El "observador" jurídico (quién cuenta como agente legítimo) es análogo al problema de la frontera cuántica.

5.10 Síntesis de demostraciones formales del capítulo [A-Meta]

Nosotros presentamos la siguiente tabla de demostraciones formales en prosa integradas en este capítulo. Estas pruebas elevan el estatus epistémico de las secciones formales de [A-Código] a [A-Meta] + [A-Código], replicando el protocolo usado en el Capítulo 6.

Designación	Enunciado	Método	Sección
T-NC-Z	La medición en Z desordena la información sobre X	Inspección de definición	5.3.1
T-NC-X	La medición en X desordena la información sobre Z	Inspección de definición	5.3.1
T-NoSignal-Z	La medición en Z preserva probabilidades marginales incompatibles	Inspección de definición	5.3.3
T-NoSignal-X	La medición en X preserva probabilidades marginales incompatibles	Inspección de definición	5.3.3
T-NoSignal-D	La medición en D preserva probabilidades marginales incompatibles	Inspección de definición	5.3.3
T-CHSH-Alg	Cota algebraica clásica CHSH: $\|S\| \leq 2$	Análisis de casos $b = b' \,/\, b \neq b'$	5.3.2 Parte A
T-CHSH-Scale	$\|classical_S\| \leq 200$	T-CHSH-Alg + factor de escala 100	5.3.2 Parte A
T-Bell-Exact	$S_{quantum} = -2\sqrt{2}$	Identidades trigonométricas	5.3.2 Parte B
T-Violation	$\|S_{quantum}\| > 2$	T-Bell-Exact + $\sqrt{2} > 1$	5.3.2 Parte C
T-Context	La ordenación de mediciones incompatibles importa	Cálculo directo	5.3.4

T-CFD-Impl	CFD implica valores simultáneos definidos	Existencias sucesivas	5.3.5
T-CFD-Block	El modelo epistémico bloquea CFD	Inspección de tupla	5.3.5
T-Z3-UNSAT	El realismo local clásico es insatisfacible	Modus ponens + dato experimental	5.4.1
T-Z3-Bohm	Viabilidad de la teoría de Bohm	Exhibición de modelo	5.4.1
T-Z3-QBism	Viabilidad de QBism / OQM	Exhibición de modelo	5.4.1
T-Z3-Superdet	Viabilidad del superdeterminismo	Exhibición de modelo	5.4.1
T-Z3-GRW	Viabilidad de GRW / colapso objetivo	Exhibición de modelo	5.4.1
T-Z3-Retro	Viabilidad de retrocausalidad	Exhibición de modelo	5.4.1
T-Z3-Relativist	Viabilidad del relativismo ontológico	Exhibición de modelo	5.4.1
T-Z3-Radical	Viabilidad del pluralismo radical	Exhibición de modelo	5.4.1
T-AS-Crisis	$crisis(1)$ pertenece a todo answer set	Restricción hard	5.4.2
T-AS-Frag	$fragmentation(2)$ pertenece a todo answer set	Restricción hard	5.4.2
T-AS-Stab	Al menos un agente estabilizado en todo answer set	Restricción hard	5.4.2

Conclusión metodológica [C]. El Capítulo 5 posee materia prima para 23 demostraciones formales en prosa. Estas pruebas son correctas, completas y verificables por un lector con formación básica en lógica, teoría de modelos elemental, Lean 4, Z3 y semántica de answer sets. Su integración junto a los códigos Lean, Z3 y Clingo eleva el estatus epistémico del capítulo de una verificación meramente computacional a una demostración formal mixta **[A-Meta] + [A-Código]**.

Etiquetado del capítulo

Sección	Contenido	Clasificación
5.0	Introducción	[C] Interpretación
5.1	Protocolo de cuatro preguntas	[C] Interpretación
5.2	Modelo epistémico (Lean 4)	[A-Código]
5.3	Teoremas estructurales (Lean 4)	[A-Meta] + [A-Código]
5.3.1	No-conmutatividad	[A-Meta] + [A-Código]
5.3.2	Frontera de Bell	[A-Meta] + [A-Código]
5.3.3	Condición de no-señalización	[A-Meta] + [C]

5.3.4	Contextualidad	[A-Meta] + [A-Código]
5.3.5	Bloqueo de CFD	[A-Meta] + [A-Código]
5.4.1	Z3: Mapa del pluralismo interpretativo	[A-Meta] + [A-Código]
5.4.2	Clingo: Dinámica del discurso cuántico	[A-Meta] + [A-Código]
5.4.3	Conexión con protocolo	[C] Interpretación
5.5	Interpretaciones	[B]
5.6	Entrelazamiento y no-localidad	[B/C]
5.6.1	Crítica realista a Bell	[B/C]
5.6.2	Renuncia ontológica	[C]
5.7	Pedagogía	[B/C]
5.8	Limitaciones	[C]
5.9	Conexiones	[C]
5.10	Síntesis de demostraciones formales	[A-Meta] + [C]

Capítulo 6: La reificación de la conjunción en la recepción del teorema de Bell: axiomática, interpretación y reificación

6.0 Introducción: Bell es correcto, pero la recepción estándar extrae una conclusión más fuerte de lo que su demostración autoriza [C]

En el Capítulo 5, nosotros formalizamos la medición cuántica como cambio de nivel epistémico y mapeamos, mediante Z3, las siete interpretaciones viables que surgen al renunciar a al menos uno de los tres supuestos metafísicos del teorema de Bell (Localidad, Definición Contracausal, Independencia Estadística). En este capítulo, nosotros aislamos y formalizamos con precisión quirúrgica la operación lógica específica que opera en la recepción pedagógica y divulgativa estándar del teorema de Bell: la inferencia de que Bell "refutó el realismo local" cuando, estrictamente, Bell refutó una conjunción específica de tres supuestos separables bajo una estructura probabilística particular.

Debemos ser absolutamente claros desde el inicio sobre el alcance de nuestra crítica:

Lo que NO cuestionamos:

El desarrollo matemático de Bell es impecable. La derivación algebraica que conduce a la cota $|S| \leq 2$ bajo la hipótesis de distribución conjunta es correcta.

Los datos experimentales de Aspect (1982), Weihs (1998) y Hensen (2015) confirman $S \approx 2{,}828$, violando esa cota [B].

Bell mismo fue cuidadoso en sus escritos (Bell 1964, 1981): su teorema refuta las teorías de variables ocultas locales bajo su definición específica [B].

Lo que SÍ cuestionamos:

La recepción pedagógica y divulgativa estándar opera mediante dos movimientos que este capítulo analiza:

Primero, la elisión axiomática. La recepción estándar trata como evidente que el realista local está forzado a formalizar los resultados de los detectores mediante una distribución de probabilidad conjunta $P(A, A', B, B')$ sobre las cuatro variables simultáneas. Esta asunción equivale, por el teorema de Fine (1982), a la Definición Contracausal (CFD): los experimentos no realizados tienen resultados definidos [B]. La recepción estándar presenta esta estructura como consecuencia necesaria del realismo local, cuando en realidad es un supuesto adicional separable.

Segundo, la totalización de la conjunción. La recepción estándar interpretó $\neg(R \wedge L \wedge DC)$ como $\neg(R \wedge L)$. Esa inferencia solo es válida si se acepta el lema adicional $(R \wedge L) \rightarrow DC$. Sin ese lema, la conclusión correcta es simplemente: "la conjunción de estos tres supuestos bajo esta estructura probabilística es insostenible", no "el realismo y la localidad son insostenibles".

Importante: Si se adopta la definición estándar de "realismo local" que Bell y EPR usaron (donde la CFD es constitutiva del realismo, no un supuesto separable), entonces el teorema de Bell SÍ refuta válidamente esa posición, y la inferencia de la recepción estándar es lógicamente correcta. Lo que este capítulo denuncia no es una falacia lógica formal en ese caso, sino una falacia pedagógica de totalización: presentar la conjunción $R \wedge L \wedge CFD$ como bloque monolítico ("el realismo local"), ocultando que CFD es un supuesto separable y que otras definiciones de realismo (como el realismo mínimo que adoptamos aquí) permiten preservar localidad sin CFD.

La reificación aquí opera exactamente como en los capítulos anteriores: se trata una conjunción de tres supuestos separables como si fuera una hipótesis unitaria e inseparable, y al refutar uno de sus componentes (CFD), se transfiere erróneamente la refutación a los otros dos (R y L).

Declaración de alcance. Este capítulo no propone un modelo físico alternativo que explique las correlaciones cuánticas desde primeros principios. Lo que hacemos es analizar la estructura lógica del argumento de Bell y de su recepción estándar, y mostrar, mediante formalización en Z3 y simulación en Clingo, que el espacio de posibilidades interpretativas es más amplio de lo que la narrativa estándar sugiere.

Nota sobre el tono. Este capítulo no "destruye" a Bell. Lo acota, y sobre todo acota a su recepción estándar. La contribución de Bell es enorme: mostró que la conjunción de esos tres supuestos bajo estructura clásica es insostenible. Nuestro aporte es mostrar que la conjunción no es una unidad indivisible, y que la recepción estándar reificó esa conjunción al presentarla como una sola hipótesis.

6.1 Aplicación del protocolo al teorema de Bell [C]

Aplicamos las cuatro preguntas del Capítulo 0 al caso del teorema de Bell y su recepción estándar.

Pregunta 1: ¿Cuál es el objeto formal y cuál es su tipo?

El objeto formal es la conjunción $R \wedge L \wedge CFD$ como estructura lógica compuesta. Su tipo real es: conjunción de tres supuestos independientes. No es una hipótesis unitaria; es una tríada separable donde cada componente puede ser afirmado o negado independientemente de los otros dos.

En el formalismo de Bell (1964), esta conjunción se expresa mediante:

Realismo (R): Las partículas poseen propiedades definidas independientemente de ser medidas. Los detectores revelan, no inventan.

Localidad (L): La elección de eje de medición en Alice no altera instantáneamente el resultado en Bob. No-señalización.

CFD (Definición Contracausal): Los experimentos no realizados tienen resultados definidos. Es decir, A y A' existen simultáneamente.

Estos tres supuestos son lógicamente independientes: se puede afirmar uno sin afirmar los otros. La conjunción es una estructura relacional entre proposiciones, no una sustancia unitaria.

Pregunta 2: ¿Qué atribuye la narrativa a este objeto que su tipo no contiene?

La narrativa estándar (recepción pedagógica y divulgativa) atribuye a la conjunción $R \wedge L \wedge CFD$ propiedades de hipótesis unitaria e inseparable:

"Bell demostró que el realismo local es falso" (totalización: trata la conjunción como bloque monolítico).

"La naturaleza es no-local" (inferencia inválida: de $\neg(R \wedge L \wedge CFD)$ se infiere $\neg L$, ignorando que la CFD podría ser el componente refutado).

"Einstein estaba equivocado" (personificación + simplificación: reduce un programa de investigación a una persona).

"No hay variables ocultas locales" (naturalización: presenta la ausencia de un tipo específico de modelo como ausencia de toda posibilidad realista).

Estas atribuciones requieren tratar la conjunción como una sustancia con identidad propia, algo que puede ser "refutado" en bloque. Pero el tipo real de la conjunción es una estructura relacional entre tres proposiciones separables. Hay un mismatch de tipo. Eso es la reificación.

Los procesos satélite que operan aquí son:

Totalización: Borrar la heterogeneidad interna de la conjunción, presentándola como "el realismo local" sin distinguir sus componentes.

Naturalización: Presentar el lema $(R \wedge L) \rightarrow CFD$ como si fuera una consecuencia lógica inevitable en lugar de una elección axiomática de la recepción estándar.

Sacralización: Elevar el teorema de Bell a un estatus intocable que bloquea la revisión de sus premisas.

Pregunta 3: ¿Existe una descripción alternativa sin ese atributo?

Sí. Existe una descripción donde la conjunción se presenta como lo que es: tres supuestos separables, cada uno con su propio estatus.

La descripción alternativa es el realismo contextual: un realismo mínimo donde los detectores no inventan los resultados (el resultado medido es real, no un artefacto del aparato), pero donde no se postulan valores simultáneos para mediciones no realizadas. Formalmente, esto equivale a:

$R =$ true (los detectores no inventan)

$L =$ true (no-señalización)

$CFD =$ false (no hay valores definidos para mediciones no realizadas)

Modelo estadístico: cuatro distribuciones separadas P_{ab}, $P_{ab'}$, $P_{a'b}$, $P_{a'b'}$, una por sub-experimento, sin distribución conjunta global.

Esta posición tiene anclaje en la literatura: Pitowsky (1989) demostró que las desigualdades de Bell son teoremas de la probabilidad clásica, no de la localidad [B]. Fine (1982) demostró la equivalencia CFD ⇔ Distribución Conjunta [B]. Spekkens (2005) desarrolló modelos contextuales que preservan realismo y localidad [B].

Pregunta 4: ¿Es teoría resuelta con narrativa sucia?

Sí. El teorema de Bell es una teoría resuelta: la derivación algebraica es correcta, las predicciones están confirmadas experimentalmente con precisión extraordinaria, y hay consenso sobre los datos. Lo que está "sucio" es la narrativa que lo envuelve en su recepción estándar: la afirmación "Bell refutó el realismo local" cuando estrictamente Bell refutó $R \wedge L \wedge CFD$.

La narrativa sucia opera mediante totalización: presenta la conjunción como si fuera una hipótesis unitaria, y al refutarla, transfiere la refutación a todos sus componentes simultáneamente. El formalismo es sólido; la interpretación estándar de la recepción es la capa reificada.

6.2 Formalización en Z3: la estructura lógica de la elisión axiomática [A-Meta] + [A-Código]

Nosotros construimos un sistema formal en Z3 que captura la estructura lógica de la elisión axiomática en la recepción estándar del teorema de Bell. El sistema no modela la física cuántica; modela la estructura argumentativa del teorema y sus premisas ocultas.

Las demostraciones en prosa [A-Meta] se presentan junto con las verificaciones computacionales [A-Código]. Las pruebas en prosa no son reemplazadas por el código; el código verifica instancias concretas y controla errores de tipeo.

6.2.1 Ontología formal: tres niveles de compromiso

El sistema distingue tres niveles de compromiso:

Nivel ontológico (compromisos sobre la naturaleza de la realidad):

```
; Realismo (R) = REALISMO MÍNIMO: los detectores revelan, no inventan.
; NO incluye CFD. Es la afirmación de que el resultado medido es real,
; no un artefacto del aparato. No afirma que las mediciones NO realizadas
; tengan valores definidos simultáneamente.
(declare-const Realism Bool)

; Localidad (L): la elección de eje de medición en Alice no altera
```

```
; instantáneamente el resultado en Bob. No-señalización.
(declare-const Locality Bool)

; Definición Contracausal (CFD): los experimentos no realizados
; tienen resultados definidos. Es decir, A y A' existen simultáneamente.
; Nota: Fine (1982) demostró que CFD ⇔ Distribución Conjunta.
(declare-const CFD Bool)

; Distribución Conjunta Clásica (DC): existe una única medida de
; probabilidad P(A, A', B, B') sobre las 4 variables simultáneas.
(declare-const JointDist Bool)
```

Nivel epistemológico (el terreno neutral del estadístico):

```
; FourSubexperiments: los datos provienen de 4 corridas físicamente
; distintas. En cada una solo se miden 2 observables, nunca 4.
(declare-const FourSubexperiments Bool)

; OnlyDetectorData: el estadístico solo tiene acceso a los registros
; de los medidores (+1/-1), no a "propiedades subyacentes".
(declare-const OnlyDetectorData Bool)
```

Nivel metodológico (elecciones del modelado estadístico):

```
; StatisticianImposesJoint: el estadístico postula una distribución
; conjunta P(A,A',B,B') como modelo subyacente.
; NOTA: esto es una ELECCIÓN DE MODELO, no una consecuencia de los datos.
(declare-const StatisticianImposesJoint Bool)

; StatisticianUsesContextual: el estadístico usa 4 distribuciones
; separadas P_ab, P_ab', P_a'b, P_a'b', una por sub-experimento.
(declare-const StatisticianUsesContextual Bool)
```

Parámetro empírico:

```
(declare-const S_value Real)
```

6.2.2 Axiomas del sistema formal

```
; AXIOMA 1 (Metodológico): Si los datos provienen de 4 experimentos
; separados y solo observamos detectores, no hay base empírica para
; postular una distribución conjunta sobre 4 variables no medidas
; simultáneamente.
(assert (! (=> (and FourSubexperiments OnlyDetectorData)
               (not StatisticianImposesJoint))
         :named A1_NoEmpiricalBasisForJoint))

; AXIOMA 2 (Elección de Bell): Bell identificó su modelo formal con
; el realismo local. Pero JointDist es una elección metodológica,
; no una consecuencia ontológica.
```

```
(assert (! (= JointDist StatisticianImposesJoint)
         :named A2_JointIsMethodologicalChoice))

; AXIOMA 3 (Teorema de Fine): CFD es matemáticamente equivalente
; a la existencia de una distribución de probabilidad conjunta.
(assert (! (= CFD JointDist)
         :named A3_Fine_Equivalence))

; AXIOMA 5 (Teorema de Bell, matemáticamente correcto):
; Si se acepta Realismo ∧ Localidad ∧ DistribuciónConjunta,
; entonces necesariamente |S| ≤ 2.
(assert (! (=> (and Realism Locality JointDist) (<= S_value 2.0))
         :named A5_CHSH_Correct))

; AXIOMA 6 (Dato experimental): Aspect, Weihs, Hensen et al.
(assert (! (= S_value 2.828) :named A6_Experimental_Data))

; POSTULADO DE VIABILIDAD (verificado algebraicamente en 6.2.4):
; Un realista local que renuncia a la Distribución Conjunta puede usar
; un modelo contextual que es lógicamente compatible con S = 2.828.
; NOTA: Este postulado se verifica como consistencia algebraica en la
; Sección 6.2.4 mediante el script contextual_model_z3.smt2.
(assert (! (=> (and Realism Locality StatisticianUsesContextual)
               (and (not JointDist) (= S_value 2.828)))
         :named A7_ContextualRealism_FitsData))

; AXIOMA 8 (Exclusión metodológica): el estadístico o impone
; distribución conjunta o usa modelo contextual, no ambos.
(assert (! (xor StatisticianImposesJoint StatisticianUsesContextual)
         :named A8_MutualExclusion))
```

6.2.3 El lema de la recepción estándar como definición, no como axioma global

Esta es la decisión metodológica más importante del script. El lema que la recepción estándar asume implícitamente —que el realismo y la localidad implican la CFD— se define como proposición pero no se aserta globalmente. Se invoca selectivamente para mostrar su efecto destructivo:

```
; La recepción estándar asume implícitamente que Realismo ∧ Localidad
; implica CFD. Por Fine (A3), esto equivale a imponer Distribución
Conjunta.
; Definimos A4 como proposición, pero NO lo asertamos globalmente.
(declare-const A4_Reception_Fallacious_Lemma Bool)
(assert (! (= A4_Reception_Fallacious_Lemma (=> (and Realism Locality)
CFD))
         :named A4_Definition))
```

Nota sobre los axiomas. Siguiendo la reflexión del Capítulo 2 (sección 2.7), nosotros reconocemos que el Axioma 1 es una elección filosófica, no un descubrimiento empírico. Se ancla en el empirismo metodológico: los datos de cuatro sub-experimentos separados no fuerzan una distribución conjunta sobre variables no medidas simultáneamente. Esta es una posición defendible pero no es una verdad lógica necesaria. El lector puede cuestionar esta elección; en ese caso, los teoremas cambian.

Nota sobre la elección de Z3. Z3 es la herramienta adecuada porque el problema es de satisfacibilidad (¿existe un modelo donde $R \wedge L$ sea consistente con $S = 2{,}828$?), no de demostración deductiva pura. Z3 produce modelos concretos o núcleos insatisfacibles, que es exactamente lo que necesitamos.

6.2.4 Verificación algebraica del modelo contextual [A-Meta] + [A-Código]

El Postulado de Viabilidad (A7) afirma que existe un modelo contextual compatible con los datos. Para verificar esto, construimos un script Z3 que verifica la consistencia algebraica de las correlaciones cuánticas.

Definición 6.2.4 (Modelo Contextual Realista-Local). Un Modelo Contextual $\mathcal{M}_{ctx}$ para el experimento EPRB-CHSH es una tupla de cuatro espacios de probabilidad clásicos separados, uno por sub-experimento:

$$\mathcal{M}_{ctx} = \{(\Omega_{ab}, \mathcal{F}_{ab}, P_{ab}, A_{ab}, B_{ab}), (\Omega_{ab'}, \mathcal{F}_{ab'}, P_{ab'}, A_{ab'}, B_{ab'}), (\Omega_{a'b}, \mathcal{F}_{a'b}, P_{a'b}, A_{a'b}, B_{a'b}), (\Omega_{a'b'}, \mathcal{F}_{a'b'}, P_{a'b'}, A_{a'}$$

donde para cada par de orientaciones $(x, y) \in \{(a, b), (a, b'), (a', b), (a', b')\}$:

$(\Omega_{xy}, \mathcal{F}_{xy}, P_{xy})$ es un espacio de probabilidad clásico.

$A_{xy}: \Omega_{xy} \to \{-1, +1\}$ y $B_{xy}: \Omega_{xy} \to \{-1, +1\}$ son variables aleatorias.

Localidad contextual: $P_{xy}(A_{xy} = u, B_{xy} = v) = P_{xy}(A_{xy} = u) \cdot P_{xy}(B_{xy} = v)$ para todo $u, v \in \{-1, +1\}$. Dentro de cada sub-experimento, los resultados son estadísticamente independientes.

Realismo mínimo: Para cada $\omega \in \Omega_{xy}$, los valores $A_{xy}(\omega)$ y $B_{xy}(\omega)$ existen independientemente de los detectores. Los detectores revelan, no inventan.

Ajuste empírico: $E_{xy} = \int_{\Omega_{xy}} A_{xy}(\omega) B_{xy}(\omega) dP_{xy}(\omega) = -\cos(\theta_x - \theta_y)$.

No existencia de distribución conjunta: No se postula ningún espacio $(\Omega, \mathcal{F}, P)$ ni variables A, A', B, B' definidas simultáneamente sobre Ω tales que las cuatro distribuciones P_{xy} sean marginales de P.

Verificación algebraica (Teorema 6.2.5). El script *contextual_model_z3.smt2* verifica que las cuatro correlaciones cuánticas predichas por la regla de Born ($E_{ab} = -\sqrt{2}/2$,

$E_{ab'} = +\sqrt{2}/2$, $E_{a'b} = -\sqrt{2}/2$, $E_{a'b'} = -\sqrt{2}/2$) son mutuamente consistentes como números reales y satisfacen $S_{ctx} = -2\sqrt{2} \approx -2{,}828$, violando la cota CHSH clásica $|S| \leq 2$.

Resultados verificados del script contextual_model_z3.smt2:

Z3 devuelve sat con el siguiente modelo algebraico:

Símbolo Z3	Expresión root-obj	Valor exacto	Significado
sqrt2	root-obj(x^2 - 2, 2)	$+\sqrt{2} \approx 1{,}4142$	La constante que definimos
E_ab	root-obj($2x^2$ - 1, 1)	$-\sqrt{2}/2 \approx -0{,}7071$	Correlación cuántica (a, b)
E_abp	root-obj($2x^2$ - 1, 2)	$+\sqrt{2}/2 \approx +0{,}7071$	Correlación cuántica (a, b')
E_apb	root-obj($2x^2$ - 1, 1)	$-\sqrt{2}/2 \approx -0{,}7071$	Correlación cuántica (a', b)
E_apbp	root-obj($2x^2$ - 1, 1)	$-\sqrt{2}/2 \approx -0{,}7071$	Correlación cuántica (a', b')
S_ctx	root-obj(x^2 - 8, 1)	$-2\sqrt{2} \approx -2{,}828$	Combinación CHSH
CHSH_bound	2.0	2,0	Cota clásica

Verificación: $|S_{ctx}| = 2\sqrt{2} \approx 2{,}828 > 2{,}0 = CHSH_bound$. ✓

Teoremas algebraicos integrados [A-Meta]

Teorema T-Alg [A-Meta]

Valor exacto de la combinación CHSH contextual.

Enunciado. Sean $E_{ab} = -\frac{\sqrt{2}}{2}$, $E_{ab'} = +\frac{\sqrt{2}}{2}$, $E_{a'b} = -\frac{\sqrt{2}}{2}$, $E_{a'b'} = -\frac{\sqrt{2}}{2}$. Defínase $S_{ctx} = E_{ab} - E_{ab'} + E_{a'b} + E_{a'b'}$. Entonces $S_{ctx} = -2\sqrt{2}$ y $|S_{ctx}| > 2$.

Demostración. Sustituimos los valores:

$$S_{ctx} = \left(-\frac{\sqrt{2}}{2}\right) - \left(+\frac{\sqrt{2}}{2}\right) + \left(-\frac{\sqrt{2}}{2}\right) + \left(-\frac{\sqrt{2}}{2}\right) = -4 \cdot \frac{\sqrt{2}}{2} = -2\sqrt{2}.$$

Como $\sqrt{2} > 1{,}414$, tenemos $|S_{ctx}| = 2\sqrt{2} > 2 \cdot 1{,}414 = 2{,}828 > 2$. ▫

Teorema T-Fine [A-Meta]

Imposibilidad de distribución conjunta en el modelo contextual.

Enunciado. Bajo Σ, si $|S_{ctx}| > 2$ y $Realism \wedge Locality$ es verdadero, entonces $\neg JointDist$ (y por $A3$, $\neg CFD$).

Demostración. Supóngase $JointDist$ verdadero. Entonces $Realism \wedge Locality \wedge JointDist$. Por $A5$, $|S| \leq 2$. Pero por T-Alg, $|S_{ctx}| = 2\sqrt{2} > 2$. Como el modelo contextual ajusta los datos ($A6$), la variable S_value toma el valor $2{,}828 = 2\sqrt{2}$. Luego

$|S_value| \leq 2$ es falso. Contradicción. Por tanto $\neg JointDist$. Por $A3$ (equivalencia), $\neg CFD$. ▫

Nota. Este teorema es la reconstrucción formal del Corolario 6.2.6. No demuestra el teorema de Fine (que es citado como [B] y postulado como $A3$), sino que usa $A3$ para derivar la consecuencia lógica dentro del sistema del capítulo.

Nota sobre el alcance de esta verificación. El script verifica la consistencia algebraica de las correlaciones cuánticas y su violación de CHSH. No construye explícitamente las distribuciones de probabilidad P_{xy} con variables aleatorias $A_{xy}(\omega), B_{xy}(\omega) \in \{-1, +1\}$ que produzcan esos valores esperados manteniendo independencia local. La construcción explícita de tales distribuciones excede el alcance lógico-epistemológico de este capítulo.

La consistencia del modelo contextual como descripción epistemológica se sostiene porque la mecánica cuántica misma calcula cada correlación por separado mediante la regla de Born sobre el estado singlete $|\Psi^-\rangle$, sin postular una distribución conjunta $P(A, A', B, B'|\Psi^-)$. Por el teorema de Fine (1982), la violación de CHSH implica que no existe una distribución conjunta clásica de la cual las cuatro correlaciones sean marginales. El modelo contextual es consistente precisamente porque renuncia a postular tal distribución.

Observación crucial. El Modelo Contextual no es una teoría física que compita con la mecánica cuántica como descripción fundamental. Es una descripción epistemológica de los datos: el estadístico que solo tiene acceso a los registros de los detectores puede describir cada sub-experimento con su propio espacio de probabilidad, sin necesidad de postular un espacio común que unifique los cuatro.

Corolario 6.2.6 (Verificación algebraica del Postulado de Viabilidad). El Postulado de Viabilidad (A7) se verifica como consistencia algebraica: el script confirma que los cuatro valores cuánticos son mutuamente consistentes como números reales y violan la cota CHSH, lo cual (por Fine 1982) implica que no requieren una distribución conjunta. El script no construye explícitamente las distribuciones P_{xy} con variables ± 1; eso excede el alcance lógico-epistemológico de este capítulo.

6.2.5 Metateoremas del sistema axiomático Z3 [A-Meta]

Para mantener la higiene epistémica, nosotros separamos explícitamente lo demostrable de lo no-demostrable con los materiales del capítulo. Todo lo que aquí enunciamos se deriva exclusivamente de los axiomas A1–A8, las definiciones 6.2.4, los hechos del programa Clingo y las reglas de agregación del capítulo. No añadimos datos empíricos, interpretaciones sociológicas ni física cuántica externa.

Lo que NO admite demostración formal [A-Meta] a partir del Capítulo 6

Enunciado del capítulo	Razón
"La recepción estándar reifica la conjunción"	Tesis sociológica/interpretativa [C]

“Fine (1982) demostró la equivalencia CFD⇔JD”	Cita de literatura [B]; en el capítulo es axioma A3
“Pitowsky demostró que Bell es teorema de probabilidad, no de localidad”	Cita [B]
“Spekkens desarrolló modelos contextuales”	Cita [B]
“La pedagogía de Bell opera mediante totalización”	Análisis discursivo [B/C]
“El realismo mínimo es una posición filosófica legítima”	Tesis filosófica [C]
“El modelo Clingo no es historia real”	Advertencia metodológica [C]
Cualquier afirmación sobre “mensajes instantáneos” o “acción a distancia”	Interpretación física [C]

Familia I: Metateoremas del sistema axiomático Z3

Teorema T-Cons [A-Meta]

Consistencia del sistema base.

Enunciado. Sea $\Sigma = \{A1, A2, A3, A5, A6, A7, A8\}$ el conjunto de axiomas del sistema formal de la sección 6.2. Entonces Σ es consistente.

Demostración. Exhibimos un modelo $\mathcal{M}$. Sea el dominio de interpretación el conjunto de valores booleanos y reales estándar. Definimos:

$Realism^{\mathcal{M}} = \top$, $Locality^{\mathcal{M}} = \top$, $CFD^{\mathcal{M}} = \bot$, $JointDist^{\mathcal{M}} = \bot$.

$FourSubexperiments^{\mathcal{M}} = \top$, $OnlyDetectorData^{\mathcal{M}} = \top$.

$StatisticianImposesJoint^{\mathcal{M}} = \bot$, $StatisticianUsesContextual^{\mathcal{M}} = \top$.

$S_value^{\mathcal{M}} = 2{,}828$.

$A4^{\mathcal{M}}$ puede definirse arbitrariamente (por ejemplo, $\bot$).

Verificación axioma por axioma:

A1: Antecedente $FourSub \wedge OnlyData$ es $\top$; consecuente $\neg StatisticianImposesJoint$ es $\neg\bot = \top$. La implicación es $\top$.

A2: $JointDist = StatisticianImposesJoint$ se satisface: $\bot = \bot$.

A3: $CFD = JointDist$ se satisface: $\bot = \bot$.

A5: Antecedente $Realism \wedge Locality \wedge JointDist = \top \wedge \top \wedge \bot = \bot$. La implicación es vacuamente $\top$.

A6: $S_value = 2{,}828$ por asignación.

A7: Antecedente $Realism \land Locality \land StatisticianUsesContextual = \top \land \top \land \top = \top$. Consecuente $\neg JointDist \land (S_v alue = 2{,}828) = \top \land \top = \top$. La implicación es $\top$.

A8: $StatisticianImposesJoint \oplus StatisticianUsesContextual = \bot \oplus \top = \top$.

Como todo axioma de Σ es verdadero en $\mathcal{M}$, Σ es satisfacible, luego consistente. ▫

Teorema T-Ind [A-Meta]

Independencia del lema A4.

Enunciado. El lema $A4$: $(Realism \land Locality) \rightarrow CFD$ es independiente de Σ. Es decir, ni $A4$ ni $\neg A4$ son teoremas de Σ.

Demostración. Debemos exhibir dos modelos de Σ que difieran en el valor de verdad de $A4$.

Modelo $\mathcal{M}_1$ donde $A4$ es verdadero: Modificamos $\mathcal{M}$ de T-Cons fijando $Realism^{\mathcal{M}_1} = \bot$. Entonces el antecedente de $A4$ es $\bot$, luego $A4$ es $\top$ en $\mathcal{M}_1$. Los demás axiomas se siguen satisfaciendo porque $A5$ y $A7$ tienen antecedentes que se vuelven $\bot$ o se mantienen $\top$ según corresponda, y los demás axiomas no dependen de $Realism$.

Modelo $\mathcal{M}_2$ donde $A4$ es falso: Tomamos $\mathcal{M}$ de T-Cons tal cual. Allí $Realism = \top$, $Locality = \top$, $CFD = \bot$. El antecedente de $A4$ es $\top$ y el consecuente $\bot$, luego $A4$ es $\bot$ en $\mathcal{M}_2$.

Como Σ tiene modelos donde $A4$ es verdadero y modelos donde es falso, $A4$ es independiente de Σ. ▫

Teorema T-NDev [A-Meta]

No-derivabilidad de la inferencia estándar.

Enunciado. $\Sigma \not\vdash (Realism \land Locality) \rightarrow JointDist$.

Demostración. Por reducción al absurdo semántico. Supóngase que la fórmula fuera teorema de Σ. Entonces sería verdadera en todo modelo de Σ. Pero en el modelo $\mathcal{M}_2$ de T-Ind (que es modelo de Σ), tenemos $Realism = \top$, $Locality = \top$, $JointDist = \bot$. Luego $(Realism \land Locality) \rightarrow JointDist$ es $\bot$ en $\mathcal{M}_2$. Contradicción. ▫

Nota. Este metateorema formaliza exactamente la "falacia estructural" que el capítulo atribuye a la recepción estándar: la inferencia de que realismo y localidad implican distribución conjunta no es válida en el sistema axiomático base.

Teorema T-Equiv [A-Meta]

La inferencia estándar ES válida bajo A4.

Enunciado. $\Sigma \cup \{A4\} \vdash \neg(Realism \land Locality)$.

Demostración. Supóngase $Realism \land Locality$. Por $A4$, se sigue CFD. Por $A3$ $(CFD \leftrightarrow JointDist)$, se sigue $JointDist$. Entonces $Realism \land Locality \land JointDist$. Por $A5$, se

sigue $S_value \leq 2$. Por $A6$, $S_value = 2{,}828$. Luego $2{,}828 \leq 2$, una contradicción aritmética. Por tanto, la suposición $Realism \wedge Locality$ es imposible en $\Sigma \cup \{A4\}$. Se sigue $\neg(Realism \wedge Locality)$. ▫

Nota. Este teorema demuestra formalmente la concesión honesta del capítulo: si se adopta la definición estándar donde $A4$ es analítico (es decir, si $Realism$ incluye CFD por definición), entonces la recepción estándar es lógicamente correcta. La crítica del capítulo no es a la validez lógica bajo esas premisas, sino a la presentación de $A4$ como inevitable cuando es una elección definitorial.

6.3 Los siete teoremas: verificación de la elisión axiomática [A-Meta] + [A-Código]

Con el sistema axiomático definido, nosotros planteamos siete teoremas que exploran el espacio de consistencia. Cada teorema corresponde a una pregunta específica sobre la estructura lógica del argumento de Bell y su recepción estándar.

Teorema 1 (UNSAT): Bell es matemáticamente correcto dentro de su marco [A-Meta] + [A-Código]

Enunciado [A-Meta]. $\Sigma \cup \{Realism, Locality, JointDist\} \vdash \bot$.

Demostración [A-Meta]. Supóngase $Realism \wedge Locality \wedge JointDist$. Por el axioma $A5$:

$$(Realism \wedge Locality \wedge JointDist) \rightarrow (S_value \leq 2).$$

Por modus ponens, $S_value \leq 2$. Por el axioma $A6$, $S_value = 2{,}828$. Luego $2{,}828 \leq 2$. Esta desigualdad aritmética es falsa. Se deriva $\bot$. ▫

Verificación computacional [A-Código]:

```
(push)
(assert (! (and Realism Locality JointDist) :named T1_Bell_Target))
(check-sat)
(get-unsat-core)
(pop)
```

Resultado verificado:

```
unsat
(T1_Bell_Target A5_CHSH_Correct A6_Experimental_Data)
```

Interpretación: Si se acepta $R \wedge L \wedge JointDist$, por A5 se obtiene $S \leq 2$, pero A6 dice $S = 2{,}828$. Contradicción. Z3 identifica el núcleo insatisfacible mínimo: exactamente tres axiomas están en tensión. Esto confirma que Bell es matemáticamente correcto dentro de su marco. Nadie refuta el álgebra.

Teorema 2 (UNSAT): La incompatibilidad surge al invocar A4 [A-Meta] + [A-Código]

Enunciado [A-Meta]. $\Sigma \cup \{A4, Realism, Locality\} \vdash \bot$.

Demostración [A-Meta]. Supóngase $A4 \wedge Realism \wedge Locality$. Por $A4$, $(Realism \wedge Locality) \rightarrow CFD$. Por modus ponens con las dos últimas premisas, CFD. Por $A3$ $(CFD \leftrightarrow JointDist)$, $JointDist$. Entonces $Realism \wedge Locality \wedge JointDist$. Por T1, esto implica $\bot$. ▫

Verificación computacional [A-Código]:

```
(push)
(assert (! A4_Reception_Fallacious_Lemma :named A4_Invoked))
(assert (! Realism :named R_T2))
(assert (! Locality :named L_T2))
(assert (! A6_Experimental_Data :named Data_T2))
(check-sat)
(get-unsat-core)
(pop)
```

Resultado verificado:

```
unsat
(L_T2 A3_Fine_Equivalence A5_CHSH_Correct A6_Experimental_Data
 A4_Definition R_T2 A4_Invoked)
```

Interpretación: Al invocar A4 (el lema $R \wedge L \rightarrow CFD$), el sistema se vuelve inconsistente. El núcleo insatisfacible incluye explícitamente *A4_Invoked*.

Nota sobre la función diagnóstica de T2. Matemáticamente, T2 es equivalente a T1 dado A3 (Fine: $CFD \Leftrightarrow JointDist$). Su función no es revelar una contradicción nueva, sino diagnosticar la etiología de la contradicción: mostrar que el núcleo insatisfacible incluye explícitamente el lema A4. Es análogo a un médico que, sabiendo que el paciente tiene fiebre, hace una prueba para confirmar que la causa es la bacteria X y no la Y.

Teorema 3 (SAT): El escape realista existe [A-Meta] + [A-Código]

Enunciado [A-Meta]. $\Sigma \cup$
$\{Realism, Locality, StatisticianUsesContextual, \neg JointDist, A6\}$ es consistente.

Demostración [A-Meta]. Exhibimos el modelo $\mathcal{M}$ de T-Cons. Allí:

$Realism = \top, Locality = \top, StatisticianUsesContextual = \top.$

$\neg JointDist = \neg\bot = \top.$

$A6$ se satisface con $S_value = 2{,}828$.

Todos los axiomas de Σ se satisfacen (por T-Cons), y las fórmulas adicionales son verdaderas en $\mathcal{M}$. Luego el conjunto es consistente. ▫

Verificación computacional [A-Código]:

```
(push)
(assert (! Realism :named R_T3))
(assert (! Locality :named L_T3))
(assert (! StatisticianUsesContextual :named SUC_T3))
(assert (! (not JointDist) :named NotJD_T3))
(assert (! A6_Experimental_Data :named Data_T3))
(check-sat)
(get-model)
(pop)
```

Resultado verificado:

```
sat
(define-fun Realism () Bool true)
(define-fun Locality () Bool true)
(define-fun StatisticianUsesContextual () Bool true)
(define-fun JointDist () Bool false)
(define-fun CFD () Bool false)
(define-fun S_value () Real (/ 707.0 250.0))
```

Interpretación: Z3 encontró un modelo concreto donde:

$Realism = true$

$Locality = true$

$JointDist = false$

$CFD = false$

$S_value = 2{,}828$

Este es el núcleo de la demostración: existe un mundo posible donde el realista local preserva ambos compromisos usando un modelo contextual. El realista no está obligado a usar distribución conjunta.

Teorema 4 (SAT): El terreno neutral no fuerza distribución conjunta [A-Meta] + [A-Código]

Enunciado [A-Meta]. $\{A1, A8\} \cup \{FourSubexperiments, OnlyDetectorData, StatisticianUsesContextual, \neg StatisticianImposesJoint\}$ es consistente.

Demostración [A-Meta]. Sea $\mathcal{M}_4$ la restricción de $\mathcal{M}$ (de T-Cons) a las variables relevantes. Allí $FourSub = \top, OnlyData = \top, SUC = \top, SIJ = \bot$. Verificamos:

$A1$: $\top \rightarrow \neg\bot = \top$.

$A8$: $\bot \oplus \top = \top$.

Las demás fórmulas son verdaderas por asignación. ▫

Verificación computacional [A-Código]:

```
(push)
(assert (! FourSubexperiments :named FS_T4))
(assert (! OnlyDetectorData :named ODD_T4))
(assert (! StatisticianUsesContextual :named SUC_T4))
(assert (! (not StatisticianImposesJoint) :named NotSIJ_T4))
(check-sat)
(get-model)
(pop)
```

Resultado verificado:

```
sat
(define-fun FourSubexperiments () Bool true)
(define-fun OnlyDetectorData () Bool true)
(define-fun StatisticianUsesContextual () Bool true)
(define-fun StatisticianImposesJoint () Bool false)
```

Interpretación: El terreno empírico neutral (4 sub-experimentos separados, solo datos de detectores) no fuerza al estadístico a postular distribución conjunta. Los datos no obligan a elegir un modelo sobre otro.

Teorema 5 (SAT): RΛL sin A4 es viable [A-Meta] + [A-Código]

Enunciado [A-Meta]. $\Sigma \cup \{Realism, Locality, A6\} \vdash \neg JointDist$.

Demostración [A-Meta]. Supóngase, para reducir al absurdo, que $JointDist$ es verdadero. Entonces $Realism \wedge Locality \wedge JointDist$. Por $A5$, $S_value \leq 2$. Pero $A6$ exige $S_value = 2{,}828$. Contradicción. Luego $\neg JointDist$. ▫

Corolario inmediato [A-Meta]. Bajo las mismas premisas, $StatisticianUsesContextual$ es verdadero y $StatisticianImposesJoint$ es falso (por $A8$).

Verificación computacional [A-Código]:

```
(push)
(assert (! Realism :named R_T5))
(assert (! Locality :named L_T5))
(assert (! A6_Experimental_Data :named Data_T5))
(check-sat)
```

```
(get-model)
(pop)
```

Resultado verificado:

```
sat
(define-fun Realism () Bool true)
(define-fun Locality () Bool true)
(define-fun JointDist () Bool false)
(define-fun CFD () Bool false)
(define-fun StatisticianUsesContextual () Bool true)
(define-fun S_value () Real (/ 707.0 250.0))
```

Interpretación: Sin el lema que la recepción estándar asumió (A4), $R \wedge L$ es perfectamente consistente con $S = 2{,}828$. La estructura axiomática fuerza $JointDist = false$ y $StatisticianUsesContextual = true$ como vía de escape.

Teorema 6 (SAT): La invalidez de la inferencia estándar [A-Meta] + [A-Código]

Enunciado [A-Meta]. El conjunto $\{\neg(Realism \wedge Locality \wedge JointDist), Realism, Locality, A6\} \cup \Sigma$ es consistente.

Demostración [A-Meta]. Exhibimos el modelo $\mathcal{M}$ de T-Cons. Evaluamos:

$\neg(R \wedge L \wedge JD) = \neg(\top \wedge \top \wedge \bot) = \neg\bot = \top$.

$Realism = \top, Locality = \top$.

$A6$ satisfecho.

Luego existe un modelo donde la conjunción de la tríada es falsa, pero $Realism$ y $Locality$ son verdaderos. Esto prueba que $\neg(R \wedge L \wedge JD)$ no implica $\neg(R \wedge L)$. ▫

Verificación computacional [A-Código]:

```
(push)
(assert (! (not (and Realism Locality JointDist)) :named NotRLJD_T6))
(assert (! Realism :named R_T6))
(assert (! Locality :named L_T6))
(assert (! A6_Experimental_Data :named Data_T6))
(check-sat)
(get-model)
(pop)
```

Resultado verificado:

```
sat
(define-fun Realism () Bool true)
(define-fun Locality () Bool true)
```

```
(define-fun JointDist () Bool false)
(define-fun S_value () Real (/ 707.0 250.0))
```

Interpretación: Aceptar $\neg(R \wedge L \wedge JD)$ y mantener $R \wedge L$ es lógicamente consistente. Esto formaliza exactamente la invalidez de la inferencia estándar: de $\neg(A \wedge B \wedge C)$ no se sigue $\neg(A \wedge B)$ a menos que se asuma $(A \wedge B) \rightarrow C$.

Nota sobre la analogía del triángulo equilátero rojo. Una forma común de ilustrar esta estructura lógica es: "Bell demostró que no existe un triángulo equilátero rojo; la recepción estándar concluyó que no existe un triángulo equilátero. La conclusión solo sigue si se asume que todo triángulo equilátero debe ser rojo."

Advertencia sobre la analogía: Esta analogía ilustra nuestra posición (donde CFD es contingente respecto al realismo), pero no prueba su corrección. Para Bell y para la definición estándar de realismo local, CFD era constitutiva del realismo (el "rojo" no era contingente sino esencial). Bajo esa definición, la inferencia de la recepción estándar es lógicamente válida. La analogía es pedagógicamente útil para mostrar la estructura de la elisión axiomática, pero la corrección de nuestra posición depende de cuál definición de realismo se adopte, no de la analogía.

Teorema 7 (SAT): Fine + rechazo de A4 = realismo local contextual viable [A-Meta] + [A-Código]

Enunciado [A-Meta]. $\Sigma \cup \{\neg A4, Realism, Locality, StatisticianUsesContextual, A6\}$ es consistente.

Demostración [A-Meta]. En el modelo $\mathcal{M}$ de T-Cons, tenemos $Realism = \top$, $Locality = \top$, $CFD = \bot$. Evaluamos $A4 = (R \wedge L) \rightarrow CFD = \top \rightarrow \bot = \bot$. Luego $\neg A4 = \top$. Los demás axiomas se satisfacen como en T-Cons. ▫

Verificación computacional [A-Código]:

```
(push)
(assert (! (not A4_Reception_Fallacious_Lemma) :named NotA4_T7))
(assert (! Realism :named R_T7))
(assert (! Locality :named L_T7))
(assert (! StatisticianUsesContextual :named SUC_T7))
(assert (! A6_Experimental_Data :named Data_T7))
(check-sat)
(get-model)
(pop)
```

Resultado verificado:

```
sat
(define-fun Realism () Bool true)
(define-fun Locality () Bool true)
(define-fun A4_Reception_Fallacious_Lemma () Bool false)
```

```
(define-fun StatisticianUsesContextual () Bool true)
(define-fun JointDist () Bool false)
(define-fun CFD () Bool false)
(define-fun S_value () Real (/ 707.0 250.0))
```

Interpretación: Aceptando la equivalencia de Fine (A3: $CFD \Leftrightarrow JointDist$) pero rechazando explícitamente el lema que la recepción estándar asumió (A4), Z3 construye un modelo donde el realista local preserva todos sus compromisos ontológicos usando modelo contextual. La consistencia algebraica de este modelo está verificada por el script *contextual_model_z3.smt2* (Sección 6.2.4), que demuestra que las cuatro correlaciones cuánticas satisfacen $S_{ctx} = -2\sqrt{2} \approx -2{,}828$ violando la cota CHSH.

Tabla resumen de los siete teoremas

Teorema	Resultado	Núcleo / Modelo clave	Interpretación
T1	unsat	T1_Bell_Target + A5_CHSH_Correct + A6_Experimental_Data	Bell es matemáticamente correcto: $R \wedge L \wedge DC$ es incompatible con $S = 2{,}828$
T2	unsat	Incluye A4_Invoked	La incompatibilidad surge al invocar el lema A4 (función diagnóstica)
T3	sat	$R = true, L = true, SUC = true, JD = false, S = 2{,}828$	El núcleo de la demostración: el realista local contextual es perfectamente viable
T4	sat	$FourSub = true, OnlyData = true, SUC = true, SIJ = false$	El terreno común no fuerza distribución conjunta
T5	sat	$R = true, L = true, JD = false, S = 2{,}828$	Sin A4, $R \wedge L$ encaja con los datos
T6	sat	$R = true, L = true, JD = false, S = 2{,}828$	Aceptar $\neg(R \wedge L \wedge DC)$ y mantener $R \wedge L$ es consistente
T7	sat	$R = true, L = true, \neg A4, SUC = true, JD = false, S = 2{,}828$	Fine + rechazo de A4 = realismo local contextual viable (consistencia algebraica verificada en 6.2.4)

La elisión axiomática en una línea

La recepción estándar demostró (correctamente): $\neg(R \wedge L \wedge DC)$.

Pero concluyó (inválidamente sin A4): $\neg(R \wedge L)$.

Esto es lógicamente inválido sin el lema $(R \wedge L) \rightarrow DC$.

Pitowsky y Fine demostraron que DC se deriva de la CFD, no de la localidad.

Por tanto, la elisión axiomática en la recepción estándar es una petición de principio estadístico: impone una estructura de probabilidad conjunta como si fuera consecuencia ontológica de la localidad, cuando es solo una elección metodológica.

Importante: Si se adopta la definición estándar de "realismo local" (donde CFD es constitutiva), entonces $R \rightarrow CFD$ es analítico, A4 es verdadero por definición, y la inferencia de la recepción estándar es lógicamente válida. La "elisión axiomática" que denunciamos es pedagógica y retórica: la totalización de la conjunción que ocultó que CFD es un supuesto separable bajo otras definiciones de realismo.

6.4 Clingo: Dinámica epistémica de la comunidad científica [A-Meta] + [A-Código]

Si el espacio de modelos de Z3 es estático —un mapa de posibilidades lógicas—, el modelo Clingo es dinámico: simula cómo una comunidad de agentes con compromisos ontológicos explícitos transitaría entre modelos estadísticos ante la evidencia.

6.4.1 Estructura del modelo

El código Clingo modela:

5 agentes con compromisos ontológicos declarados:

locality_first_realists (comprometidos con R + L)

realism_first_realists (comprometidos con R)

bellian_orthodox (comprometidos con R + L, pero usan joint_model)

epistemic_pragmatists (comprometidos con L)

experimental_neutrals (ningún compromiso ontológico previo)

4 modelos estadísticos con propiedades declaradas:

joint_model: requiere CFD, no ajusta $S = 2{,}828$

contextual_model: no viola localidad ni realismo, ajusta $S = 2{,}828$

epistemic_model: viola realismo, ajusta $S = 2{,}828$

bohm_model: viola localidad, ajusta $S = 2{,}828$

Restricciones de coherencia ontológica:

Un agente comprometido con localidad NO puede adoptar un modelo que viole localidad.

Un agente comprometido con realismo NO puede adoptar un modelo que viole realismo.

Optimización multiobjetivo:

Prioridad 1 (peso 4): minimizar costo ontológico (satélites activos)

Prioridad 2 (peso 3): minimizar uso de joint_model post-t=0

Prioridad 3 (peso 2): maximizar ajuste empírico

Prioridad 4 (peso 1): minimizar violaciones de compromiso

Prioridad 5 (peso 0): minimizar cambios bruscos

6.4.2 Resultado verificado (OPTIMUM FOUND, Answer 23)

```
Optimization: 5 0 -15 6
OPTIMUM FOUND
```

Métrica	Valor	Significado
Costo ontológico	5	Solo satélites heredados de t=0. Post-crisis: cero entidades no medidas postuladas
Uso de joint_model	0	Nadie usa el modelo de Bell después de t=0
Ajuste empírico	-15	Máximo absoluto: los 5 agentes ajustan los datos en los 3 tiempos
Cambios	6	Mínimo necesario para escapar del paradigma conjunto

---

6.4.3 Trayectoria de cada agente en el óptimo

Agente	Compromisos	t=0	t=1	t=2	t=3	Destino final
locality_first_realists	L + R	joint	contextual	contextual	contextual	Única opción viable
bellian_orthodox	L + R	joint	contextual	contextual	contextual	Única opción viable
realism_first_realists	R	joint	contextual	contextual	contextual	Preferido sobre Bohm

epistemic_pragmatists	L	joint	contextual	epistemic	epistemic	Contextual como puente
experimental_neutrals	(ninguno)	joint	bohm	bohm	bohm	Libre elección

6.4.4 Interpretación del resultado

- El realista local es forzado al modelo contextual. *locality_first_realists* y *bellian_orthodox* tienen compromisos con ambos principios (R y L). Las restricciones de coherencia los bloquean:

✗ *bohm_model* → viola localidad (acción a distancia)

✗ *epistemic_model* → viola realismo (no hay valores preexistentes)

✗ *joint_model* → inconsistente con datos (requiere CFD)

✓ *contextual_model* → única opción restante

2. El modelo contextual es preferido incluso por realistas puros. *realism_first_realists* (solo comprometidos con realismo, podrían aceptar no-localidad) igual eligen *contextual_model* sobre *bohm_model*. El optimizador penaliza el costo ontológico: Bohm postula una ontología más extravagante (onda piloto no-local). El contextual es más parsimonioso.

3. El paradigma de la recepción estándar es históricamente inestable. Cero agentes permanecen en *joint_model* post-t=0. La ortodoxía de la recepción estándar no es un atractor del sistema. La evidencia de 4 sub-experimentos + el teorema de Fine la desestabilizan irreversiblemente.

4. Fragmentación post-crisis con un núcleo contextual. Aunque la comunidad se fragmenta (*fragmentation(2,3)*), el *contextual_model* emerge como el atractor dominante: 3 de 5 agentes se estabilizan allí.

Nota sobre el estatus de esta simulación. El modelo Clingo muestra qué ocurriría bajo ciertos criterios de racionalidad (parsimonia ontológica, coherencia, ajuste empírico). No es una reconstrucción histórica de cómo evolucionó realmente el debate sobre Bell. La historia real involucra factores sociológicos, institucionales y económicos que el modelo no captura.

Conexión con el protocolo anti-reificación: La simulación demuestra que si la comunidad científica operara con compromisos ontológicos explícitos y parsimonia (minimizar entidades no medidas postuladas), la interpretación contextual emergería naturalmente como la opción dominante para los realistas locales —sin necesidad de renunciar ni al realismo ni a la localidad. La reificación de la conjunción $R \wedge L \wedge CFD$ como "el realismo local" es lo que impide ver esta tercera vía.

6.4.5 Metateoremas formales de la dinámica Clingo [A-Meta]

Teorema T-Coh [A-Meta]

Coherencia ontológica fuerza el modelo contextual.

Enunciado. En el programa Clingo de la sección 6.4, sea a un agente cuyos compromisos ontológicos incluyan $Realism$ y $Locality$. Sea AS un answer set que satisfaga las restricciones de coherencia ontológica del programa. Entonces para todo tiempo $t > 0$: $adopta(a, contextual_model, t) \in AS$, y $adopta(a, M, t) \notin AS$ para $M \in \{bohm_model, epistemic_model, joint_model\}$.

Demostración. Análisis de casos exhaustivo sobre los cuatro modelos disponibles.

Caso 1: $M = bohm_model$. El programa declara que $bohm_model$ viola $Locality$. La restricción de coherencia ontológica del programa elimina (mediante regla de restricción *:-*) todo answer set donde un agente comprometido con $Locality$ adopte $bohm_model$. Luego $adopta(a, bohm_model, t) \notin AS$.

Caso 2: $M = epistemic_model$. El programa declara que $epistemic_model$ viola $Realism$. Análogamente, la restricción de coherencia elimina esta opción para agentes comprometidos con $Realism$. Luego $adopta(a, epistemic_model, t) \notin AS$.

Caso 3: $M = joint_model$. El programa asocia a $joint_model$ la propiedad de requerir CFD (equivalente a $JointDist$ por $A3$). Por la dinámica del programa y las weak constraints de ajuste empírico, un agente que adopte $joint_model$ en $t > 0$ incurre en costo por no ajustar los datos (o directamente viola restricciones de optimalidad). Más aún, el answer set óptimo verificado tiene costo 0 en joint_model post-t=0, lo cual implica que ningún agente lo adopta. Luego $adopta(a, joint_model, t) \notin AS$ en el óptimo.

Caso 4: $M = contextual_model$. Este modelo no viola $Realism$ ni $Locality$ según la declaración del programa, y ajusta los datos. No está bloqueado por ninguna restricción de coherencia.

Como la choice rule del programa fuerza a cada agente a adoptar exactamente un modelo, y los casos 1–3 son imposibles, necesariamente $adopta(a, contextual_model, t) \in AS$. ▫

Teorema T-Opt [A-Meta]

Optimalidad lexicográfica del answer set 23.

Enunciado. Sea AS_{23} el answer set con vector de costo $(5{,}0, -15{,}6)$ reportado en la sección 6.4.2. Entonces AS_{23} es óptimo bajo el orden lexicográfico de weak constraints del programa.

Demostración. Análisis por prioridades.

Prioridad 2 (peso 3, minimizar uso de joint_model post-t=0): El valor 0 significa que ningún agente adopta $joint_model$ en $t > 0$. Este es el mínimo absoluto posible, ya que el costo es una suma de penalizaciones no negativas. Cualquier answer set con valor $>$ 0 en esta prioridad es estrictamente dominado por AS_{23}.

Prioridad 3 (peso 2, maximizar ajuste empírico): El valor -15 corresponde a 5 agentes $\times$ 3 tiempos ($t = 1,2,3$) de ajuste perfecto, cada uno con costo -1. Este es el mínimo (más negativo) posible. Cualquier answer set con valor > -15 es estrictamente dominado en esta prioridad.

Como AS_{23} alcanza los óptimos absolutos en las prioridades 2 y 3, ningún answer set puede dominarlo lexicográficamente. Las prioridades 1 y 4 solo desempatizan entre answer sets que empatan en 2 y 3; por tanto, AS_{23} es globalmente óptimo. ▫

Teorema T-NoAtr [A-Meta]

Inestabilidad del joint_model.

Enunciado. En la dinámica definida por el programa Clingo, $joint_model$ no pertenece a ningún answer set óptimo en tiempo $t > 0$.

Demostración. Por la weak constraint de prioridad 2, cada instancia de $adopta(A, joint_model, t)$ con $t > 0$ contribuye con costo estrictamente positivo al vector de optimización. Sin embargo, por T-Opt, existe un answer set (AS_{23}) con costo 0 en esta prioridad. Por definición de optimalidad lexicográfica en Clingo, cualquier answer set con costo mayor en la prioridad más alta dominante es estrictamente peor. Luego ningún answer set que contenga $joint_model$ en $t > 0$ puede ser óptimo. ▫

Teorema T-Frag [A-Meta]

Fragmentación epistémica en el óptimo.

Enunciado. En AS_{23}, para $t = 3$, existen al menos dos agentes a_1, a_2 tales que los modelos adoptados son distintos.

Demostración. Inspección de la trayectoria reportada en la sección 6.4.3:

$locality_first_realists$ adopta $contextual_model$ en $t = 3$.

$experimental_neutrals$ adopta $bohm_model$ en $t = 3$.

Sean $a_1 = locality_first_realists$ y $a_2 = experimental_neutrals$. Entonces $adopta(a_1, contextual_model, 3) \in AS_{23}$ y $adopta(a_2, bohm_model, 3) \in AS_{23}$. Como $contextual_model \neq bohm_model$, la comunidad está fragmentada. El programa deriva el átomo $fragmentation(2,3)$ porque el conteo de modelos distintos en $t = 3$ es mayor o igual que 2. ▫

6.5 Evidencia y anclaje en la literatura [B]

La tesis de este capítulo no es una invención del autor. Tiene anclaje en la filosofía de la física contemporánea. Presentamos estos autores como antecedentes de nuestra formalización, no como posiciones a refutar.

Fine (1982): Arthur Fine demostró que las desigualdades de Bell son condiciones necesarias y suficientes para la existencia de una distribución de probabilidad conjunta sobre todas las variables de medición involucradas [B]. La equivalencia $CFD \Leftrightarrow JointDist$ es el fundamento matemático de nuestra formalización (Axioma 3).

Pitowsky (1989): Itamar Pitowsky demostró que las desigualdades de Bell son teoremas de la probabilidad clásica, no de la localidad [B]. La derivación de la cota CHSH requiere exclusivamente la existencia de una distribución conjunta; la localidad no aparece en la demostración algebraica. Esto confirma que el verdadero supuesto insostenible es la CFD, no la localidad.

Spekkens (2005): Robert Spekkens desarrolló modelos de variables ocultas contextuales que preservan realismo y localidad [B]. Estos modelos demuestran que existe un espacio lógico para un realismo local sin CFD.

Werner y Griffiths: Reinhard Werner (OQM) y Robert Griffiths (Historias Consistentes) argumentan que la violación de las desigualdades de Bell no ataca directamente la localidad, sino que demuestra que la mecánica cuántica es una teoría no-clásica [B]. La ruptura es con la estructura probabilística clásica, no con la localidad per se.

Bell (1964, 1981): En el paper original, Bell formula su "vital assumption" como la condición de localidad: "the result B for particle 2 does not depend on the setting $\vec{b}$ of the magnet for particle 1, nor A on $\vec{b}$" [B]. Sin embargo, la estructura de la ecuación (2) —$P(\vec{a},\vec{b}) = \int d\lambda, \rho(\lambda)A(\vec{a},\lambda)B(\vec{b},\lambda)$— ya presupone que A y B están definidos para todo λ, lo cual es CFD implícita [B]. Bell fue cuidadoso en sus propios escritos al referirse a "teorías de variables ocultas locales" bajo su definición específica. La elisión axiomática que este capítulo denuncia es de la recepción estándar que simplificó su resultado a "Bell refutó el realismo local".

Invariant Set Theory (Palmer): Tim Palmer propone que ciertas mediciones no pueden realizarse simultáneamente debido a principios nomológicos que restringen el conjunto de estados físicamente posibles [B]. Esto ofrece una vía para violar la Independencia Estadística sin caer en superdeterminismo, mediante una "restricción contracausal" donde el contrafactual es físicamente imposible en lugar de simplemente desconocido.

Nota sobre el estatus epistémico. Estos autores proporcionan anclaje [B] para la tesis central. Nuestra contribución no es descubrir algo nuevo, sino hacer explícito con Z3 lo que ellos intuyeron en prosa: que la conjunción $R \wedge L \wedge CFD$ no es una unidad indivisible, y que el realista puede renunciar a CFD sin renunciar a R ni a L.

6.6 La pedagogía del teorema de Bell: cómo se enseña la totalización [B/C]

La investigación en educación física (PER) y el análisis de libros de texto revelan que la reificación de la conjunción de Bell no es un accidente; es una estrategia pedagógica sistemática [B].

Frases típicas de libros de texto

Frase estándar	Operación de reificación
"Bell demostró que el realismo local es falso"	Totalización: trata la conjunción como bloque monolítico
"La naturaleza es no-local"	Inferencia inválida: de $\neg(R \wedge L \wedge CFD)$ se infiere $\neg L$
"Einstein estaba equivocado"	Personificación + simplificación
"No hay variables ocultas locales"	Naturalización: presenta la ausencia de un tipo específico de modelo como ausencia de toda posibilidad realista
"El entrelazamiento implica acción a distancia"	Reificación de la correlación como mecanismo causal

El proceso pedagógico estándar

Se presenta $R \wedge L$ como bloque unitario ("el realismo local").

Se deriva la desigualdad CHSH.

Se muestra la violación experimental ($S = 2{,}828$).

Se concluye "el realismo local es falso" (sin distinguir qué componente cayó).

La alternativa pedagógica

Presentar R, L y CFD como supuestos separables.

Derivar la desigualdad.

Mostrar la violación.

Concluir: "la conjunción de estos tres supuestos es insostenible; elija cuál abandonar".

Mostrar que existen 7 interpretaciones viables (como en la sección 5.4.1 del Capítulo 5).

La paradoja pedagógica

La reificación es necesaria para el aprendizaje inicial: presentar "el realismo local" como bloque permite al estudiante entender el teorema sin manejar tres supuestos simultáneamente. Pero la reificación es perniciosa si no se desactiva: produce la misconception persistente de que "Bell refutó el realismo" cuando solo refutó una versión específica que incluye CFD.

La solución es la misma que proponemos en el Capítulo 4: usar la reificación como prótesis cognitiva, pero hacer visible la prótesis. Enseñar con la simplificación, pero señalar que es una simplificación.

Conexión con Edelstein

En la entrevista del 27 de julio de 2026 en el programa "El Maravilloso Mundo de la Física" (Gelatina), el físico teórico José Edelstein dice sobre el entrelazamiento: "Tiene que haber un mensaje instantáneo… resulta que efectivamente ocurre esto" [B]. Esta frase reifica la correlación estadística como mecanismo causal ("mensaje"). El propio Edelstein, en la misma entrevista, invoca la metáfora de Borges sobre el mapa y el territorio, mostrando que sabe que el formalismo no postula "mensajes". La reificación opera como prótesis pedagógica inevitable, no como error por ignorancia.

6.7 El debate filosófico: realismo mínimo vs. realismo-CFD [C]

Debemos abordar honestamente la objeción principal: ¿tiene derecho el autor a llamar "realista" a una posición que niega la CFD?

La definición estándar (Bell/EPR)

En la tradición de Bell, "realismo" se define como: las partículas poseen valores definidos para todas las magnitudes medibles, independientemente de si se miden o no. Bajo esta definición, la CFD es parte constitutiva del realismo, y A4 se vuelve analítico (verdadero por definición). Si se adopta esta definición, la recepción estándar no puede ser cuestionada lógicamente: el realismo local, así definido, implica CFD, y CFD implica distribución conjunta, y distribución conjunta implica $|S| \leq 2$.

Consecuencia importante: Bajo la definición estándar de realismo local (la de Bell, que incluye CFD como constitutiva), el teorema de Bell SÍ refuta válidamente esa posición, y no hay falacia lógica en la inferencia de la recepción estándar. La "elisión axiomática" que denunciamos es pedagógica y retórica: la totalización de la conjunción como bloque monolítico, que ocultó que CFD es un supuesto separable y que otras definiciones de realismo (como el realismo mínimo) permiten preservar localidad sin CFD.

La definición mínima (este libro)

Nosotros adoptamos una definición más débil de realismo: los detectores no inventan los resultados; miden algo que es independiente del medidor. Bajo esta definición:

El resultado que obtuve no fue inventado por el aparato.

No afirmo que todos los resultados que podría haber obtenido ya estaban definidos simultáneamente.

"El resultado que obtuve no fue inventado" ≠ "Todos los resultados posibles ya estaban definidos".

Esta distinción es real y filosóficamente legítima. Un termómetro mide la temperatura real de una habitación (realismo mínimo) sin que eso implique que la temperatura de todas las habitaciones posibles del universo está definida simultáneamente.

La objeción y la respuesta

Objeción: Si se define "realismo" como incluyendo CFD por definición, entonces A4 es analítico y la recepción estándar no puede ser cuestionada lógicamente.

Respuesta: Correcto. Pero esa es una elección terminológica, no un hecho lógico. La pregunta es: ¿es la definición estándar la única legítima? Nosotros sostenemos que no. La definición mínima (detectores no inventan) es una posición filosófica coherente y defendible. No es la posición que Bell atacó; Bell atacó el realismo-CFD bajo su definición específica. Pero eso no la vuelve ilegítima.

Posición del libro

Adoptamos "realismo contextual" como posición legítima, citando a Spekkens y la tradición de realismo estructural [B]. No pretendemos que sea la única definición de realismo; pretendemos mostrar que la definición estándar no es la única posible, y que la elección de definición no puede resolver el debate por sí sola.

Concesión honesta: Un defensor de la definición estándar puede rechazar nuestro "realismo mínimo" como insuficiente. Esto es legítimo. Nuestro punto no es que la definición estándar sea incorrecta, sino que no es la única, y que la narrativa "Bell refutó el realismo" es ambigua porque no especifica qué definición de "realismo" está usando.

6.8 Limitaciones [C]

Para completar el análisis, nosotros señalamos explícitamente lo que este capítulo no puede hacer.

Primera limitación: Las variables booleanas son simplificaciones.

"Realismo", "localidad" y "CFD" tienen formulaciones mucho más matizadas en la literatura. El modelo captura la estructura lógica, no toda la riqueza del debate. El nivel de abstracción es adecuado para el propósito (mostrar la estructura de la elisión axiomática), pero no modela la física completa.

Segunda limitación: El modelo contextual es una descripción epistemológica, no una teoría física.

El script *contextual_model_z3.smt2* verifica la consistencia algebraica de las correlaciones cuánticas; no proporciona un modelo físico completo de variables contextuales que derive las correlaciones cuánticas desde principios más fundamentales. El MC es una descripción epistemológica, no una teoría física alternativa. La pregunta "¿por qué las correlaciones tienen los valores cuánticos?" sigue abierta para el realista contextual; la respuesta es "porque así lo dicta la mecánica cuántica mediante la regla de Born", pero eso es aceptar la MC como descripción, no como explicación fundamental.

Tercera limitación: A1 es una elección filosófica, no un teorema.

La afirmación "4 sub-experimentos separados no fuerzan distribución conjunta" es defendible pero no es una verdad lógica necesaria. Un defensor de la distribución conjunta puede argumentar que la CFD es una consecuencia ontológica del realismo, no una elección metodológica. Esta objeción es legítima; nuestro punto es que la elección debe ser explícita, no implícita.

Cuarta limitación: No resolvemos el problema físico.

No proponemos un mecanismo que explique las correlaciones cuánticas desde primeros principios. Solo mostramos que la inferencia estándar en la recepción del teorema de Bell es más débil de lo que se presenta.

Quinta limitación: La definición de "realismo" es disputada.

Un defensor de la definición estándar puede rechazar nuestro "realismo mínimo" como insuficiente. Esto es legítimo; nuestro punto es que la elección de definición no puede resolver el debate por sí sola.

Sexta limitación: El modelo Clingo es una simulación normativa, no historia real.

Muestra qué pasaría bajo ciertos criterios de racionalidad (parsimonia, coherencia ontológica); no predice ni explica la historia real de la recepción de Bell. La historia real involucra factores sociológicos, institucionales y económicos que el modelo no captura.

Séptima limitación: No modelamos poder coercitivo ni dinámicas institucionales.

La razón por la cual la narrativa estándar persiste a pesar de las críticas de Fine y Pitowsky involucra factores de poder académico, inercia institucional y pedagogía que trascienden la lógica formal.

6.9 Conexiones con otros capítulos [C]

Capítulo	Conexión
Cap 0 (Protocolo)	Aplicación directa de las 4 preguntas al teorema de Bell y su recepción
Cap 1	La conjunción $R \wedge L \wedge CFD$ tratada como sustancia unitaria =

(Reificación)	totalización (proceso satélite identificado en 1.2.1)
Cap 2 (Herramientas)	Z3 para satisfacibilidad (sección 2.3), Clingo para dinámica (sección 2.4); justificación de la elección
Cap 3 (SPA/Tarski)	Análogo: SPA estratifica niveles para evitar autorreferencia; aquí estratificamos supuestos para evitar inferencias inválidas. El lema A4 es el análogo normativo de la autorreferencia: una premisa que se aplica a sí misma sin justificación
Cap 4 (Gramática)	La reificación gramatical: "el realismo local es falso" trata una conjunción como sujeto unitario de un predicado. La gramática exige un sujeto sustancial para el verbo "ser refutado", aunque el formalismo describa una conjunción separable
Cap 5 (Medición cuántica)	Nota conectiva importante: El Capítulo 5 mapeó 7 interpretaciones viables usando un cubo de 8 vértices (combinaciones de L, C, I). El vértice ($L = true, C = false, I = true$) se etiquetó allí como "QBism / OQM / Copenhague estricto" (todos epistémicos, no realistas). El modelo contextual de este capítulo ocupa ese mismo vértice con realismo recuperado. ¿Por qué no aparecía en el cubo del Capítulo 5? Porque el cubo asumía implícitamente que renunciar a CFD implica renunciar al realismo —precisamente la elisión axiomática que este capítulo desmonta. El modelo contextual es el vértice ($L = true, C = false, I = true$) con realismo mínimo preservado, una posibilidad que el cubo del Capítulo 5 no capturaba porque operaba bajo la definición estándar de realismo.
Cap 7 (Normativo)	Análogo estructural: la "soberanía popular" como conjunción reificada de decisiones individuales. El Teorema de Incompetencia Reflexiva (Cap 7) es el análogo normativo de A4: un lema que se impone sin demostración

Conexión con la tesis central del libro

La elisión axiomática en la recepción estándar del teorema de Bell es un caso paradigmático de reificación de una conjunción lógica. El discurso trata $R \wedge L \wedge CFD$ como si fuera una sola cosa ("el realismo local"), y al refutar uno de los componentes (CFD), transfiere la refutación a los otros dos (R y L). Esto es exactamente la estructura del protocolo: atribuir a una estructura relacional (conjunción) propiedades de sustancia (hipótesis unitaria inseparable).

La des-reificación consiste en restaurar la transitividad canónica:

Oración reificada: "Bell refutó el realismo local."

Oración des-reificada: "Bell demostró que la conjunción de realismo, localidad y definición contracausal bajo estructura probabilística clásica es insostenible; el realista puede preservar realismo y localidad renunciando a la definición contracausal."

Conexión con la renuncia ontológica (Cap 5, sección 5.6.2)

El Capítulo 5 identificó un patrón: el progreso científico ocurre cuando renunciamos a buscar "causas ocultas" para fenómenos que tienen explicaciones estructurales. El caso de Bell es una instancia de este patrón: la intuición exige una distribución conjunta (una "tabla de valores" oculta) para explicar las correlaciones. La renuncia ontológica aquí es renunciar a esa tabla, aceptando que las correlaciones cuánticas no requieren una distribución conjunta clásica.

Fenómeno	Intuición (causa oculta)	Física (estructura manifiesta)
Movimiento	Impetus (fuerza motriz)	Simetría de traslación (Noether)
Gravedad	Fuerza gravitacional	Geometría curvada (Einstein)
Entrelazamiento	Variables ocultas / acción a distancia	No-separabilidad (Hilbert)
Correlaciones Bell	Distribución conjunta (CFD)	Modelo contextual (4 distribuciones separadas)

6.10 Síntesis de demostraciones formales del capítulo [A-Meta]

Nosotros presentamos la siguiente tabla de demostraciones formales en prosa integradas en este capítulo. Estas pruebas elevan el estatus epistémico de las secciones 6.2–6.4 de [A-Código] a [A-Meta] + [A-Código], replicando el protocolo del Capítulo 8.

Designación	Enunciado	Método	Sección
T-Cons	Consistencia del sistema base Σ	Exhibición de modelo	6.2.5
T-Ind	Independencia de A4 respecto a Σ	Dos modelos	6.2.5
T-NDev	$\Sigma \not\vdash (R \land L) \rightarrow JD$	Reducción al absurdo semántico	6.2.5
T-Equiv	$\Sigma \cup \{A4\} \vdash \neg(R \land L)$	Cadena de implicaciones + T1	6.2.5
T1	Inconsistencia de $R \land L \land JD$	Modus ponens + aritmética	6.3
T2	Inconsistencia con A4 invocado	Modus ponens + T1	6.3
T3	Existencia del escape contextual	Exhibición de modelo	6.3
T4	Neutralidad epistemológica	Exhibición de modelo	6.3
T5	$\Sigma \cup \{R, L, A6\} \vdash \neg JD$	Reducción al absurdo	6.3
T5-Cor	Bajo $\Sigma \cup \{R, L, A6\}$, se sigue SUC y $\neg SIJ$	A2 + A8	6.3
T6	Invalidez de la inferencia estándar	Exhibición de modelo	6.3
T7	Viabilidad con $\neg A4$	Exhibición de modelo	6.3

T-Alg	$S_{ctx} = -2\sqrt{2}$ y $\|S_{ctx}\| > 2$	Sustitución algebraica directa	6.2.4
T-Fine	$\|S_{ctx}\| > 2 \wedge R \wedge L \Rightarrow \neg JD$	Reducción al absurdo + A3/A5	6.2.4
T-Coh	Coherencia fuerza contextual para $R \wedge L$	Análisis de casos (4 modelos)	6.4.5
T-Opt	Optimalidad de AS_{23}	Análisis de costos lexicográficos	6.4.5
T-NoAtr	$joint_model$ no es atractor	Dominancia lexicográfica	6.4.5
T-Frag	Fragmentación en $t = 3$	Inspección constructiva	6.4.5

Conclusión metodológica [C]. El Capítulo 6 posee materia prima para las demostraciones formales en prosa que ahora aparecen integradas junto a los outputs de Z3/Clingo. Estas pruebas son correctas, completas y verificables por un lector con formación básica en lógica proposicional, teoría de modelos elemental y análisis de optimización lexicográfica.

Etiquetado del capítulo

Sección	Contenido	Clasificación
6.0	Introducción	[C] Interpretación
6.1	Protocolo de cuatro preguntas	[C] Interpretación
6.2	Formalización Z3 (axiomas y metateoremas)	[A-Meta] + [A-Código]
6.2.4	Verificación algebraica del modelo contextual	[A-Meta] + [A-Código]
6.2.5	Metateoremas del sistema axiomático Z3	[A-Meta]
6.3	Siete teoremas Z3	[A-Meta] + [A-Código]
6.4	Clingo: dinámica epistémica	[A-Meta] + [A-Código]
6.4.5	Metateoremas de la dinámica Clingo	[A-Meta]
6.5	Literatura (Fine, Pitowsky, Spekkens, Bell)	[B] Citada de fuente
6.6	Pedagogía de Bell	[B/C]
6.7	Debate filosófico (realismo mínimo vs. CFD)	[C] Interpretación
6.8	Limitaciones	[C] Interpretación
6.9	Conexiones	[C] Interpretación
6.10	Síntesis de demostraciones formales	[A-Meta] + [C]

Resumen de códigos verificados en este capítulo

Herramienta	Código	Resultado	Verificado

Z3	T1: $R \wedge L \wedge JD$	unsat + unsat core	✓
Z3	T2: $A4 \wedge R \wedge L$	unsat + unsat core (incluye A4_Invoked)	✓
Z3	T3: $R \wedge L \wedge SUC \wedge \neg JD$	sat con modelo concreto	✓
Z3	T4: $FourSub \wedge OnlyData \wedge SUC$	sat	✓
Z3	T5: $R \wedge L \wedge S = 2{,}828$	sat con $JD = false$	✓
Z3	T6: $\neg(R \wedge L \wedge JD) \wedge R \wedge L$	sat	✓
Z3	T7: $\neg A4 \wedge R \wedge L \wedge SUC$	sat	✓
Z3	Modelo contextual (Def. 6.2.4)	sat con $S_{ctx} = -2\sqrt{2}$, violación CHSH (verificación algebraica)	✓
Clingo	Dinámica epistémica (5 agentes, 4 modelos, 3 tiempos)	OPTIMUM FOUND, contextual como atractor	✓

Apéndice: Script contextual_model_z3.smt2

```
; ============================================================
; Z3: VERIFICACIÓN ALGEBRAICA DEL MODELO CONTEXTUAL
; ============================================================
; NOTA: Este script verifica la consistencia algebraica de las
; correlaciones cuánticas y su violación de CHSH. No construye
; explícitamente distribuciones de probabilidad P_xy con variables
; aleatorias A_xy(w), B_xy(w) en {+1,-1}. Eso excede el alcance
; de esta verificación formal.
; ============================================================

(set-option :produce-models true)

; ------------------------------------------------------------
; 1. sqrt(2) como constante real no interpretada
; ------------------------------------------------------------
(declare-const sqrt2 Real)
(assert (= (* sqrt2 sqrt2) 2.0))
(assert (> sqrt2 0.0))

; ------------------------------------------------------------
; 2. CORRELACIONES CUÁNTICAS para cada sub-experimento
; ------------------------------------------------------------
(declare-const E_ab Real)   ; E(a,b)   = -cos(0 - pi/4)     = -sqrt2/2
(declare-const E_abp Real)  ; E(a,b')  = -cos(0 - 3pi/4)    = +sqrt2/2
(declare-const E_apb Real)  ; E(a',b)  = -cos(pi/2 - pi/4)  = -sqrt2/2
(declare-const E_apbp Real) ; E(a',b') = -cos(pi/2 - 3pi/4) = -sqrt2/2
```

```
; Asignación de valores exactos
(assert (= E_ab    (- (/ sqrt2 2.0))))
(assert (= E_abp   (/ sqrt2 2.0)))
(assert (= E_apb   (- (/ sqrt2 2.0))))
(assert (= E_apbp  (- (/ sqrt2 2.0))))

; ------------------------------------------------------------
; 3. COMBINACIÓN CHSH EN EL MODELO CONTEXTUAL
; ------------------------------------------------------------
(declare-const S_ctx Real)
(assert (= S_ctx (+ (- E_ab E_abp) (+ E_apb E_apbp))))

; Verificamos que S_ctx = -2*sqrt2
(assert (= S_ctx (- (* 2.0 sqrt2))))

; ------------------------------------------------------------
; 4. VIOLACIÓN DE LA COTA CHSH (Teorema de Fine)
; ------------------------------------------------------------
(declare-const CHSH_bound Real)
(assert (= CHSH_bound 2.0))

; Demostramos que S_ctx < -2 (violación por debajo)
(assert (< S_ctx (- CHSH_bound)))

; ------------------------------------------------------------
; 5. VERIFICACIÓN DE CONSISTENCIA
; ------------------------------------------------------------
(check-sat)
(get-model)
```

Resultado verificado: *sat* con modelo algebraico exacto donde *S_ctx = -2√2 ≈ -2.828*, violando la cota *|S| ≤ 2*. Esto demuestra que las cuatro correlaciones cuánticas, tratadas como objetos matemáticos independientes (no como marginales de una distribución conjunta), son consistentes entre sí. Por el teorema de Fine (1982), la violación implica que no existe una distribución conjunta *P(A,A',B,B')* que las genere como marginales.

Capítulo 7: Sistemas Normativos y Ciencias Sociales

Nota metodológica sobre el estilo, la agencia y el estatus epistémico

El lector notará que la prosa de este capítulo evita deliberadamente ciertas abreviaturas comunes en el discurso académico, jurídico y científico. Con frecuencia, en lugar de escribir "el sistema sanciona", "la ley exige", "el modelo demuestra" o "el habitus presiona", nosotros escribimos "los agentes institucionales, aplicando el sistema, sancionan", "los legisladores, mediante la ley, exigen", "nosotros, mediante el modelo, verificamos" o "los periodistas, constreñidos por el habitus, omiten".

Esta falta de economía lingüística es intencional y constituye la Regla de Oro de este libro. Como demostramos en el Capítulo 1, la compresión de información mediante la nominalización, la voz pasiva y la supresión del agente es el mecanismo gramatical primario mediante el cual se produce la reificación y se diluye la responsabilidad (accountability). Nosotros nos negamos a economizar palabras si el costo de esa compresión es la pérdida del agente humano, que es el único ente con capacidad real de acción, creencia y responsabilidad moral. Herramientas como Z3 o Clingo no "demuestran" ni "deciden"; son instrumentos que nosotros operamos y cuyos resultados nosotros interpretamos. El lector debe soportar la longitud de nuestras oraciones como el precio de nuestra higiene ontológica.

Asimismo, debemos ser radicalmente transparentes sobre el estatus epistémico de nuestros teoremas formales (T1, T2, T3). Nosotros no presentamos estos teoremas como "descubrimientos empíricos" de leyes sociales inmutables. Su función no es revelar cómo funciona el mundo de manera independiente a nuestro marco teórico. Su función es operar como auditorías de coherencia axiomática. Lo que nosotros demostramos es que si los agentes aceptan las premisas sociológicas y jurídicas que hemos axiomatizado (ancladas en las tradiciones de Bourdieu, Kelsen y Luhmann), entonces la reificación, la inmunidad sistémica y la estratificación son consecuencias lógicas ineludibles y estructuralmente coherentes, no contradicciones. Un modus tollens formalizado no es un fallo metodológico; es la herramienta exacta para verificar que nuestra teoría social no contiene antinomias ocultas.

7.0 Introducción: la reificación en el discurso normativo [C]

Los capítulos anteriores de este libro han recorrido un arco que va desde la lógica formal hasta la física cuántica, pasando por la gramática del discurso científico. En cada caso, nosotros aplicamos el mismo protocolo: identificar una construcción relacional que los hablantes tratan gramaticalmente como sustancia, analizar las consecuencias de esa operación y, cuando ha sido posible, formalizar la estructura lógica del error para exhibir sus límites. En el Capítulo 4, nosotros mostramos cómo los divulgadores y pedagogos reifican el espacio-tiempo de la relatividad general; en el Capítulo 5, cómo la narrativa estándar trata la actualización epistémica de la función de onda como un "colapso" físico; en el Capítulo 6, cómo la recepción pedagógica del

teorema de Bell totaliza una conjunción de condiciones locales en una "realidad" que ninguna de ellas sostiene individualmente. En todos los casos, los hablantes aplican la misma operación gramatical: una red de relaciones adquiere, mediante un acto lingüístico, el estatus ontológico de una sustancia con propiedades propias.

En este capítulo, nosotros aplicamos ese mismo protocolo al dominio de los sistemas normativos: el derecho, el Estado, la nación, la soberanía popular. La pregunta guía que nos formulamos es directa: ¿cuándo los agentes humanos dejan de describir sus propias construcciones institucionales como redes de relaciones y comienzan a tratarlas como "sustancias" con voluntad, intención y capacidad de exigir sacrificio? [C]

El alcance de este capítulo es deliberadamente restringido. Nosotros no formalizamos sistemas jurídicos completos, ni pretendemos ofrecer una teoría política sustantiva, ni mucho menos una posición ideológica. Lo que nosotros hacemos es una auditoría gramatical y formal del discurso normativo. El código Z3 y Clingo que presentamos no es ornamental: es el microscopio que nos permite revelar estructuras que el lenguaje natural —y la teoría política tradicional en prosa— no pueden demostrar por sí solos [A].

Una nota sobre el tono. Este no es un capítulo de teoría política partidista. Si el lector espera una defensa o un ataque al Estado, a la nación o a la soberanía popular, no lo encontrará aquí. Lo que nosotros demostramos es que los conceptos de "voluntad del pueblo" y "soberanía del Estado" sufren de las mismas falacias de totalización y elisión axiomática que la recepción estándar del teorema de Bell, y que la disposición al sacrificio patriótico obedece a un cálculo lógico (Σ_R) que los propios agentes desactivan mediante la ilustración crítica [C].

7.1 Genealogía histórico-filosófica de la reificación social [B/C]

Esta sección no es una historia de la filosofía política. Es una genealogía del mismo error categorial que nosotros identificamos en física y en lógica, mostrando que la operación gramatical —tratar una relación como sustancia— es idéntica en todos los dominios. Nosotros organizamos esta genealogía por tipo de operación, no cronológicamente [B].

7.1.1 La sustancialización del poder: de Aristóteles a Hobbes [B]

Una línea hegemónica de pensadores políticos ha reificado el poder desde la antigüedad. En la Política de Aristóteles, la polis no es presentada como una convención humana sino como una sustancia natural; la comunidad política es tratada como una entidad con un telos propio, anterior a los individuos que la componen [B]. Siglos más tarde, los teólogos medievales intensificaron esta sustancialización mediante la doctrina de los "dos cuerpos del rey" (Kantorowicz, 1957). Los juristas y teólogos trataban al monarca simultáneamente como una persona física mortal y como una persona mística inmortal, encarnación de una sustancia política trascendente [B].

Thomas Hobbes representa un punto de inflexión. En el Leviathan (1651), Hobbes no presenta al Estado como una sustancia natural ni como una emanación divina, sino como una persona artificial creada por pacto. Hobbes es el primer teórico político que sabe que está reificando y que lo hace deliberadamente como tecnología política. En Hobbes, la reificación deja de ser un error involuntario para convertirse en una tecnología de poder consciente. El protocolo de este libro la sigue auditando como error categorial porque el mismatch de tipo persiste (tratar una red de pactos como un agente unitario), aunque los gobernantes lo exploten intencionalmente como una herramienta de gobierno para cohesionar a la multitud en una sola voluntad [B/C].

7.1.2 Hegel, Marx y Lukács: la ontología fantasmal [B]

Hegel eleva la reificación a una ontología del Espíritu: los filósofos hegelianos tratan al Estado como "la realidad de la Idea ética". Esta operación consta de idealización, totalización y sacralización [B]. Marx, en El Capital (1867), identifica una forma de reificación estructural: el fetichismo de la mercancía. Marx demuestra cómo los agentes económicos perciben las relaciones sociales entre personas como si fueran relaciones entre cosas [B]. La limitación del análisis de Marx es que ancla la reificación al modo de producción capitalista. Nuestro protocolo, en cambio, sostiene que los hablantes aplican la misma operación en física, en lógica y en derecho, independientemente del régimen económico [B/C].

György Lukács, en Historia y conciencia de clase (1923), extiende el análisis: los agentes bajo el capitalismo experimentan la reificación como la ontología total. Lukács describe cómo "una relación entre personas adquiere el carácter de una cosa y así adquiere una 'objetividad fantasmal'" [B]. Esta descripción es, léxicamente, idéntica a lo que nuestro protocolo formaliza: el mismatch de tipo que nuestro cálculo Σ_R captura [B/C].

7.1.3 Bourdieu, Anderson y Searle: la maquinaria de estabilización [B]

Pierre Bourdieu sistematiza la maquinaria de estabilización con los conceptos de violencia simbólica y misrecognition (desconocimiento). Bourdieu demuestra que la dominación requiere que los agentes dominados no reconozcan la arbitrariedad de la construcción institucional. Esta observación empírica tiene una contrapartida formal exacta en nuestro Teorema T1: si los agentes perciben la arbitrariedad, la reificación falla [B]. Benedict Anderson (1983) muestra que los líderes y los intelectuales construyen la nación como una comunidad imaginada, utilizando tecnologías de estabilización (censos, mapas, museos) [B]. John Searle (1995) formaliza el mecanismo lingüístico: los hablantes utilizan actos de habla declarativos ("X cuenta como Y en el contexto C") para institucionalizar la reificación [B].

7.2 Aplicación del protocolo al discurso normativo [C]

Nosotros aplicamos las cuatro preguntas del Capítulo 0 al discurso normativo:

Pregunta 1: ¿Cuál es el objeto formal y cuál es su tipo? El objeto formal es el sistema normativo, entendido como una red de normas, agentes e instituciones. Su

tipo real es: estructura relacional. Los agentes humanos no crean una "sustancia" llamada Estado; crean una red de relaciones jurídicas, administrativas y coercitivas entre personas [C].

Pregunta 2: ¿Qué atribuye la narrativa a este objeto que su tipo no contiene? Los hablantes (políticos, juristas, ciudadanos) atribuyen a esta red propiedades de voluntad unitaria, intencionalidad y capacidad de acción. Ejemplos cotidianos: "el Estado decide", "la Corte quiere", "el pueblo exige", "el mercado reaccionó". En cada caso, los hablantes sustancializan una red de decisiones individuales como si fuera un agente único con propósitos propios [C].

Pregunta 3: ¿Qué descripciones alternativas existen? Existen teorías jurídicas y políticas donde los autores tratan el derecho como sistema relacional sin reificar una entidad supra-individual. Hans Kelsen postula la norma básica como un postulado lógico, no como una sustancia. Niklas Luhmann describe sistemas autopoiéticos que procesan información sin un "sujeto" central. Michel Foucault analiza redes de poder sin centro ni agente. H.L.A. Hart define la regla de reconocimiento como una práctica social convergente, no como una entidad metafísica [C].

Pregunta 4: ¿Es teoría resuelta con narrativa sucia? Sí. Las instituciones normativas funcionan empíricamente; los agentes las operan y las predicciones sobre su comportamiento son estables. Lo que está "sucio" es la narrativa sustancialista que los propios operadores jurídicos y políticos utilizan para describir su propia práctica [C].

Dominio (Capítulos 4-6)	Objeto formal (Tipo real)	Atribución indebida (Narrativa)	Análogo normativo (Cap. 7)
Espacio-tiempo (Cap. 4)	Red de relaciones métricas	Sustancia elástica que "vibra"	El Estado como persona moral que "decide"
Función de onda (Cap. 5)	Herramienta predictiva	Entidad ontológica que "colapsa"	La Nación como entidad histórica que "exige"
Teorema de Bell (Cap. 6)	Conjunción de condiciones	Realidad totalizada	"Soberanía popular" como conjunción reificada

7.3 La gramática de la reificación: himnos, juramentos y poetas [B/C]

Esta sección conecta con la Sección 1.2.3 del Capítulo 1 (roles temáticos y supresión del agente). Nosotros analizamos himnos nacionales, juramentos oficiales y poesía patriótica como corpus empírico de la gramática de la reificación. Los líderes políticos y los intelectuales no "se equivocan" al reificar; ellos actúan como arquitectos conscientes de la operación [B].

Himnos nacionales como actos de habla compromisivos

Los himnos no son descripciones estéticas; son actos de habla compromisivos que las instituciones obligan a los ciudadanos a repetir desde la infancia para internalizar obligaciones deónticas.

Himno	Fórmula clave	Operación reificadora (Agente humano)	Proceso satélite
Argentina	"O juremos con gloria morir"	Los líderes invierten la muerte biológica en ganancia ontológica.	Sacralización
Cuba	"Morir por la patria es vivir"	Los poetas presentan el sacrificio como trascendencia.	Sacralización + Idealización
Uruguay	"¡La Patria o la tumba!"	Los legisladores imponen un falso dilema deóntico.	Totalización
Italia	"L'Italia chiamò"	Los intelectuales personifican a la entidad abstracta dotándola de voz.	Personificación
Francia	"Qu'un sang impur abreuve nos sillons"	Los revolucionarios naturalizan biológicamente la violencia territorial.	Naturalización

Juramentos oficiales: la transición del Rey a la Patria

El juramento es el acto de habla mediante el cual las instituciones vinculan al agente individual a la entidad reificada. La transición histórica es reveladora:

Juramento al Rey: Los vasallos juraban a una persona concreta, deponible, juzgable.

Juramento a la Patria: "¿Juráis a la Patria defenderla hasta perder la vida?" / "Si así no lo hiciere, que la Patria me lo demande".

Las instituciones modernas reemplazaron a la persona concreta del Rey por una abstracción reificada. Los líderes políticos utilizan a la Patria como un testigo simbólico indeponible para instalar obligaciones deónticas a gran escala. Es un cambio de tipo encubierto, idéntico al que nosotros analizamos en el discurso físico [B/C].

Los poetas como ingenieros de la reificación

Los poetas que diseñaron el andamiaje afectivo del Estado-nación moderno (Herder, Whitman, Carducci, Martí, Péguy) son ingenieros conscientes de la operación. Herder utiliza la metáfora del Volksgeist (la savia orgánica) para naturalizar la nación; Whitman fusiona químicamente la sangre del ciudadano con el suelo; Péguy sacraliza la trinchera como "tierra carnal". Los himnos y juramentos son, en términos de nuestro cálculo Σ_R, algoritmos de instalación de los axiomas de reificación que los agentes ejecutan en sus propias mentes [B/C].

7.4 Formalización del Sistema Formal de Coherencia Discursiva (SFCD) en Z3 [A]

Nosotros construimos un modelo formal en Z3 que captura la estructura de los sistemas normativos y la lógica de la reificación. El código está escrito en SMT-LIB2. Note el uso de aserciones de dominio (*distinct*) que nosotros incluimos para garantizar que los modelos generados por el solver sean pedagógicamente limpios y que los agentes sean entidades diferenciadas [A].

```
; ============================================================
; CAPÍTULO 7 — SISTEMAS NORMATIVOS Y CIENCIAS SOCIALES
; Bloque Z3: Σ_R + SFCD (Teoremas T1, T2, T3)
; ============================================================
(set-option :produce-unsat-cores true)
(set-option :produce-models true)

; 1. SORTS PRIMITIVOS
(declare-sort Agente)
(declare-sort Entidad)
(declare-sort Estrato)
(declare-sort Norma)

; 2. CONSTANTES DEL UNIVERSO
(declare-const a_critic  Agente)      ; Agente crítico (ilustrado)
(declare-const soberano  Agente)      ; Agente con poder constituido
(declare-const w         Agente)      ; Whistleblower (denunciante
interno)
(declare-const e_patria  Entidad)     ; Entidad reificada (Patria)
(declare-const s0        Estrato)     ; Estrato supremo (constitucional)
(declare-const s1        Estrato)     ; Estrato inferior (ley ordinaria)
(declare-const n_auto    Norma)       ; Norma de autolimitación
(declare-const n_violada Norma)       ; Norma que el sistema viola
(declare-const n_meta    Norma)       ; Metanorma de lealtad
institucional

; Identidad: universo poblado y distinto
(assert (! (distinct a_critic soberano w) :named distinct-agentes))
(assert (! (distinct s0 s1) :named distinct-estratos))
(assert (! (distinct n_auto n_violada n_meta) :named distinct-normas))

; 3. PREDICADOS DE Σ_R (Cálculo de Reificación)
(declare-fun CreeArbitrario (Agente Entidad) Bool)
(declare-fun Reificado    (Entidad Agente) Bool)
(declare-fun Sagrado      (Entidad Agente) Bool)
(declare-fun Sacrifica    (Agente Entidad) Bool)

; 4. PREDICADOS DE ESTRATIFICACIÓN NORMATIVA (Anclaje: SPA Cap. 3 /
Kelsen)
(declare-fun Competente (Agente Estrato) Bool)
```

```
(declare-fun Regula     (Agente Norma Estrato) Bool)
(declare-fun Evalua     (Agente Norma Estrato) Bool)
(declare-fun Menciona   (Norma Agente Estrato) Bool)

; 5. PREDICADOS DE INMUNIDAD SISTÉMICA (Anclaje: Luhmann / Bourdieu)
(declare-fun DenunciaExt  (Agente Norma) Bool)
(declare-fun DenunciaRec  (Agente Norma) Bool)
(declare-fun Interno      (Agente) Bool)
(declare-fun Delito       (Agente) Bool)
(declare-fun Sancion      (Agente) Bool)
(declare-fun Valida       (Norma) Bool)
(declare-fun Reforma      (Norma) Bool)
(declare-fun Metanorma    (Norma) Bool)

; 6. HECHOS DE DOMINIO (Background Facts)
(assert (! (Competente soberano s0) :named dom-comp-s0))
(assert (! (not (Competente soberano s1)) :named dom-no-comp-s1))
(assert (! (not (Competente a_critic s0)) :named dom-critic-no-s0))
(assert (! (not (Competente w s0)) :named dom-w-no-s0))
(assert (! (Interno w) :named dom-w-interno))
(assert (! (not (Interno a_critic)) :named dom-critic-externo))
(assert (! (not (Interno soberano)) :named dom-soberano-externo))
(assert (! (Metanorma n_meta) :named dom-n-meta))
(assert (! (Reificado e_patria soberano) :named dom-reif-soberano))
(assert (! (Reificado e_patria w) :named dom-reif-w))
(assert (! (not (Reificado e_patria a_critic)) :named dom-no-reif-
critic))

; 7. AXIOMAS DE Σ_R
; sig-A3: Los agentes reificadores bloquean su propia percepción de
arbitrariedad (Lukács)
(assert (! (forall ((x Agente) (y Entidad))
  (=> (Reificado y x) (not (CreeArbitrario x y)))) :named sig-A3-
reificacion))

; sig-A4: Los agentes sacralizan lo que previamente han reificado
(Bourdieu)
(assert (! (forall ((x Agente) (y Entidad))
  (=> (Sagrado y x) (Reificado y x))) :named sig-A4-sagrado))

; sig-A6: Los agentes solo se sacrifican por entidades que ellos
consideran sagradas
(assert (! (forall ((x Agente) (y Entidad))
  (=> (Sacrifica x y) (Sagrado y x))) :named sig-A6-sacrificio-requiere-
sagrado))

; 8. AXIOMA DE INCOMPETENCIA REFLEXIVA ESTRATIFICADA (Kelsen / Tarski)
(assert (! (forall ((a Agente) (n Norma) (s Estrato))
  (=> (and (Regula a n s) (Menciona n a s)) (not (Evalua a n s))))
  :named sig-IR-estratificacion))
```

```
; 9. AXIOMAS DE INMUNIDAD SISTÉMICA (Luhmann)
(assert (! (forall ((x Agente) (n Norma))
  (=> (and (DenunciaExt x n) (Interno x)) (Delito x))) :named T3-
metanorma-lealtad))
(assert (! (forall ((x Agente)) (=> (Delito x) (Sancion x))) :named T3-
sancion-derivada))
(assert (! (forall ((x Agente) (n Norma))
  (=> (and (DenunciaExt x n) (Interno x)) (not (DenunciaRec x n))))
  :named T3-bloqueo-reconocimiento))
(assert (! (forall ((n Norma)) (=> (not (Reforma n)) (Valida n)))
  :named T3-presuncion-vigencia))
```

7.5 Teoremas estructurales del sistema normativo y síntesis de Σ_R (Z3) [A]

En esta sección nosotros presentamos los teoremas estructurales del SFCD y del cálculo Σ_R. Cada resultado se presenta primero como una demostración matemática en prosa [A-Meta] y luego como una verificación computacional [A-Código]. Esta doble presentación garantiza que el lector pueda reconstruir la deducción sin depender exclusivamente del solver, y que el solver certifique la ausencia de errores de tipeo en los axiomas.

7.5.0 Preliminares formales: lemas de Σ_R

Antes de abordar los teoremas principales, nosotros establecemos lemas auxiliares sobre la estructura interna de Σ_R. Estos lemas capturan la cadena deóntica que vincula la reificación con el sacrificio.

Lema 1.1 (Transitividad de la cadena deóntica) [A-Meta]

Enunciado. Sea $\mathcal{L}_\Sigma$ el lenguaje de Σ_R. Entonces:

$$\Sigma_R \vdash \forall x \in Agente, \forall y \in Entidad: Sacrifica(x,y) \rightarrow Reificado(y,x).$$

Demostración. Sean $x \in Agente$ y $y \in Entidad$ arbitrarios. Supóngase $Sacrifica(x,y)$. Por el axioma $sig-A6$:

$$Sacrifica(x,y) \rightarrow Sagrado(y,x).$$

Luego $Sagrado(y,x)$. Por el axioma $sig-A4$:

$$Sagrado(y,x) \rightarrow Reificado(y,x).$$

Por tanto, $Reificado(y,x)$. ▫

Esquema de prueba. Cadena de implicaciones: $Sacrifica \xrightarrow{\text{A6}} Sagrado \xrightarrow{\text{A4}} Reificado$.

7.5.1 Teorema 1: Inmunidad del Crítico (T1)

Enunciado general (Teorema 1.1): Si un agente cree que una entidad es arbitraria, ese agente no puede sacrificarse por ella.

$$\Sigma_R \vdash \forall x \in Agente, \forall y \in Entidad: CreeArbitrario(x,y) \rightarrow \neg Sacrifica(x,y).$$

Demostración en prosa [A-Meta]:

Sean x, y arbitrarios. Supóngase, para derivar una contradicción, que $CreeArbitrario(x,y) \wedge Sacrifica(x,y)$.

Por Lema 1.1, de $Sacrifica(x,y)$ se sigue $Reificado(y,x)$.

Por el axioma $sig-A3$:

$$Reificado(y,x) \rightarrow \neg CreeArbitrario(x,y).$$

Luego $\neg CreeArbitrario(x,y)$.

Pero la hipótesis de reducción al absurdo afirma $CreeArbitrario(x,y)$. Se deriva $CreeArbitrario(x,y) \wedge \neg CreeArbitrario(x,y)$, una contradicción en Σ_R.

Por tanto, la conjunción es imposible y vale $CreeArbitrario(x,y) \rightarrow \neg Sacrifica(x,y)$. ▫

Corolario 1.2 (Instanciación al agente crítico) [A-Meta]:

$$CreeArbitrario(a_critic, e_patria) \rightarrow \neg Sacrifica(a_critic, e_patria).$$

Demostración. Instanciación universal de T1.1 con $x := a_critic, y := e_patria$. ▫

Teorema 1.3 (Inconsistencia del crítico sacrificado) [A-Meta]:

$$\Sigma_R \cup \{CreeArbitrario(a_critic, e_patria), Sacrifica(a_critic, e_patria)\} \vdash \bot.$$

Demostración. Supóngase la teoría consistente. Por Corolario 1.2, de $CreeArbitrario(a_critic, e_patria)$ se sigue $\neg Sacrifica(a_critic, e_patria)$. Pero el axioma contextual (hipótesis de la prueba) incluye $Sacrifica(a_critic, e_patria)$. Contradicción. ▫

Nota. Este es el análogo exacto en prosa del output `unsat` que Z3 certifica a continuación. El `unsat core` de Z3 exhibe precisamente las fórmulas que esta demostración invoca: `dom-no-reif-critic` (que activa la cadena negativa), `sig-A4`, `sig-A6` y la hipótesis de sacrificio.

Verificación computacional [A-Código]:

```
(push)
  (assert (! (CreeArbitrario a_critic e_patria) :named T1-hipotesis-
critico))
  (assert (! (Sacrifica a_critic e_patria)     :named T1-hipotesis-
sacrificio))
```

```
  (check-sat)
  (get-unsat-core)
(pop)
```

Output certificado: *unsat (dom-no-reif-critic sig-A4-sagrado sig-A6-sacrificio-requiere-sagrado T1-hipotesis-sacrificio)*

Interpretación: Cuando nosotros sometemos esta conjunción a Z3, el solver devuelve *unsat*. El *unsat core* activa la cadena completa de Σ_R: el agente crítico no tiene reificada a la Patria (hecho de dominio) → por lo tanto, no la considera sagrada (*sig-A4*) → por lo tanto, no puede sacrificarse por ella (*sig-A6*). La movilización sacrificial no es una opción que el crítico "rechaza" psicológicamente; es una propiedad ontológica que su propio estado epistémico le impide lógicamente derivar [A].

7.5.2 Teorema 2: Paradoja de la Soberanía / Incompetencia Reflexiva (T2)

Enunciado general (Teorema 2.1): Un agente soberano no puede emitir una norma que mencione su propia competencia y validarla en el mismo estrato sin generar contradicción.

$$Regula(a, n, s) \land Menciona(n, a, s) \land Evalua(a, n, s) \rightarrow \bot.$$

Demostración en prosa [A-Meta]:

Supónganse las tres premisas. Por el axioma $sig - IR - estratificacion$:

$$\forall a \forall n \forall s: (Regula(a, n, s) \land Menciona(n, a, s)) \rightarrow \neg Evalua(a, n, s).$$

Instanciando con los valores dados, y por modus ponens con las premisas $Regula$ y $Menciona$, se obtiene $\neg Evalua(a, n, s)$.

Pero la tercera premisa afirma $Evalua(a, n, s)$. Luego $\bot$. ▫

Corolario 2.2 (Necesidad de estrato superior) [A-Meta]:

Si un agente a regula una norma n que menciona a a, y el sistema exige que n sea evaluada, entonces la evaluación debe realizarse en un estrato $s' \neq s$.

Demostración. Sea s el estrato donde $Regula(a, n, s) \land Menciona(n, a, s)$. Por T2.1, $\neg Evalua(a, n, s)$. Si el sistema normativo postula que toda norma regulada debe ser evaluada (condición de cierre), entonces existe s' tal que $Evalua(a, n, s')$. Como $\neg Evalua(a, n, s)$, necesariamente $s' \neq s$. Si además la relación de estratos es un preorden jerárquico donde la evaluación de una norma en s requiere un meta-estrato, entonces $s \prec s'$. ▫

Nota. Este corolario es la reconstrucción formal de por qué Kelsen necesitó postular la Grundnorm en un estrato superior, y por qué Schmitt invocó un Poder Constituyente externo.

Verificación computacional [A-Código]:

```
(push)
  (assert (! (Regula soberano n_auto s0)   :named T2-regula-auto))
  (assert (! (Menciona n_auto soberano s0) :named T2-menciona-auto))
  (assert (! (Evalua soberano n_auto s0)    :named T2-evalua-auto))
  (check-sat)
  (get-unsat-core)
(pop)
```

Output certificado: *unsat(T2-menciona-auto sig-IR-estratificacion T2-regula-auto T2-evalua-auto)*

Interpretación: Este *unsat* es el análogo normativo exacto del Lema de Tarski (Capítulo 3). Los agentes no pueden derivar la validez de una norma utilizando una regla que esa misma norma establece en el mismo estrato. Hans Kelsen resolvió esto postulando la Grundnorm en un estrato *s1*; Carl Schmitt lo resolvió invocando un Poder Constituyente externo. Nosotros demostramos formalmente por qué ambos teóricos necesitaban postular un "afuera" del sistema para que los agentes pudieran evaluarlo [A].

7.5.3 Teorema 3: Inmunidad Sistémica / Whistleblower (T3a y T3b)

Enunciado general (Teorema 3.1 — Colapso del whistleblower):

$$DenunciaExt(w, n_violada) \land DenunciaRec(w, n_violada) \land Interno(w) \rightarrow \bot.$$

Demostración en prosa [A-Meta]:

Supónganse las tres premisas. Por el hecho de dominio $dom - w - interno$, tenemos $Interno(w)$. Por la hipótesis, $DenunciaExt(w, n_violada)$. Aplicando el axioma $T3 - bloqueo - reconocimiento$:

$$\forall x \forall n: (DenunciaExt(x, n) \land Interno(x)) \rightarrow \neg DenunciaRec(x, n).$$

Instanciando $x := w$, $n := n_violada$, y por modus ponens, se obtiene $\neg DenunciaRec(w, n_violada)$.

Pero la hipótesis incluye $DenunciaRec(w, n_violada)$. Contradicción. ▫

Enunciado (Teorema 3.2 — Absorción institucional forzada):

Bajo los axiomas de inmunidad sistémica:

$$DenunciaExt(w, n_violada) \land Interno(w) \land \neg Reforma(n_violada) \rightarrow Sancion(w) \land Valida(n_violada).$$

Demostración en prosa [A-Meta]:

Supónganse las premisas. Por $T3 - metanorma - lealtad$ y las dos primeras premisas, $Delito(w)$. Por $T3 - sancion - derivada$, $Delito(w) \rightarrow Sancion(w)$. Luego $Sancion(w)$.

Por la tercera premisa $\neg Reforma(n_violada)$ y el axioma $T3 - presuncion - vigencia$, $\neg Reforma(n_violada) \rightarrow Valida(n_violada)$. Luego $Valida(n_violada)$. ▫

Verificación computacional — Escenario de absorción (T3a) [A-Código]:

```
(push)
  (assert (! (DenunciaExt w n_violada) :named T3-hecho-denuncia))
  (assert (! (not (Reforma n_violada))  :named T3-absorcion-no-reforma))
  (check-sat)
  (get-model)
(pop)
```

Output certificado: *sat*. El modelo muestra *DenunciaRec(w, n_violada) = false*, *Sancion(w) = true*, *Valida(n_violada) = true*.

Nota sobre artefactos de Z3: El solver asigna valores arbitrarios (como *Sancion(x)=true* para todo el dominio) a predicados no restringidos por las hipótesis activas para minimizar el modelo. Lo relevante son las asignaciones forzadas por la lógica: los agentes institucionales, al aplicar la metanorma de lealtad que ellos mismos sostienen, no niegan la verdad de la denuncia; simplemente los agentes deciden no procesarla como tal, sino que la reclasifican como un delito para preservar la coherencia de la institución a la que pertenecen [A].

Verificación computacional — Escenario de colapso (T3b) [A-Código]:

```
(push)
  (assert (! (DenunciaExt w n_violada) :named T3-hecho-denuncia-b))
  (assert (! (DenunciaRec w n_violada) :named T3-reconocimiento-forzado))
  (assert (! (not (Reforma n_violada)) :named T3-sin-reforma))
  (check-sat)
  (get-unsat-core)
(pop)
```

Output certificado: *unsat(dom-w-interno T3-bloqueo-reconocimiento T3-reconocimiento-forzado T3-hecho-denuncia-b)*.

Interpretación: Si nosotros forzamos al modelo a exigir que los agentes institucionales reconozcan la denuncia del interno sin que exista un estrato superior que intervenga, Z3 devuelve *unsat*. La única salida lógica es que un observador externo (una revolución, una comisión de la verdad, un tribunal internacional) obligue a los agentes a reformar la norma [A].

7.5.4 Cierre de Σ_R: caracterización de la no-movilización

Los siguientes lemas y teorema establecen formalmente que la no-reificación es condición suficiente para bloquear toda la cadena deóntica del sacrificio.

Lema 1.4 (Cierre negativo de la sacralización) [A-Meta]:

$$\forall x \forall y: \neg Reificado(y, x) \rightarrow \neg Sagrado(y, x).$$

Demostración. Contrapositiva lógica del axioma $sig - A4$. ▫

Lema 1.5 (Cierre negativo del sacrificio) [A-Meta]:

$$\forall x \forall y: \neg Sagrado(y, x) \rightarrow \neg Sacrifica(x, y).$$

Demostración. Contrapositiva lógica del axioma $sig - A6$. ▫

Teorema 1.6 (Caracterización de la no-movilización) [A-Meta]:

$$\forall x \forall y: \neg Reificado(y, x) \rightarrow \neg Sacrifica(x, y).$$

Demostración. Por Lema 1.4, $\neg Reificado(y, x) \rightarrow \neg Sagrado(y, x)$. Por Lema 1.5, $\neg Sagrado(y, x) \rightarrow \neg Sacrifica(x, y)$. Por transitividad de la implicación material, se sigue el teorema. ▫

Interpretación: Este metateorema demuestra formalmente que la no-reificación es condición suficiente para bloquear toda la cadena deóntica: si un agente no ha reificado a la entidad, no puede, bajo ninguna derivación en Σ_R, llegar al sacrificio. Es la versión general del Teorema T1.

Síntesis conceptual de Σ_R

Los axiomas `sig-A3`, `sig-A4` y `sig-A6` conforman el Cálculo de Reificación (Σ_R). El estado epistémico de los agentes (su creencia consciente en la arbitrariedad histórica de la entidad) funciona como la condición lógica que bloquea la derivación del sacrificio [A]. El misrecognition (desconocimiento de la arbitrariedad) que Bourdieu identifica no es un efecto secundario de la dominación; es su condición de posibilidad lógica. Si los agentes creyeran que la Patria es arbitraria, el axioma A3 haría imposible que ellos la sacralizaran (A4), y el axioma A6 haría imposible que ellos se sacrificaran. Los poetas, los pedagogos y los líderes políticos son, en términos de Σ_R, programadores que instalan software ontológico: su función es mantener `CreeArbitrario(a,e)` en estado falso para la población objetivo [A/B].

7.6 Agencia emergente, totalización y el Teorema de Imposibilidad (Clingo + Z3) [A]

Si Z3 nos permite verificar restricciones lógicas, Clingo (Answer Set Programming) nos permite simular cómo emerge la agencia institucional a partir de reglas de agregación y cómo los agentes optimizan su estabilidad estructural. En esta sección, cada escenario se presenta primero como demostración combinatoria en prosa [A-Meta] y luego como verificación computacional [A-Código].

7.6.1 Escenario A: Tribunal Colegiado (Dilema Discursivo)

Teorema 4.1 (Doctrinal Paradox / Dilema discursivo del tribunal) [A-Meta]:

Enunciado. En el Escenario A de Clingo, sea AS el answer set del programa. Entonces:

$$conflicto \in AS.$$

Demostración. Sean j_1, j_2, j_3 los jueces y p, q, r las propuestas, con $r \leftrightarrow (p \wedge q)$.

Paso 1 (Aceptación de p). Los votos son $vota(j_1, p, 1)$, $vota(j_2, p, 1)$, $vota(j_3, p, 0)$. La regla de aceptación institucional exige 2 votos afirmativos. Luego $acepta_inst(tribunal, p) \in AS$.

Paso 2 (Aceptación de q). Los votos son $vota(j_1, q, 1)$, $vota(j_2, q, 0)$, $vota(j_3, q, 1)$. Por la misma regla, $acepta_inst(tribunal, q) \in AS$.

Paso 3 (Obligación de r). La regla $obligacion(tribunal, r) :- acepta_inst(tribunal, p), acepta_inst(tribunal, q)$ se satisface por los pasos 1 y 2. Luego $obligacion(tribunal, r) \in AS$.

Paso 4 (Rechazo de r). Los votos son $vota(j_1, r, 1)$, $vota(j_2, r, 0)$, $vota(j_3, r, 0)$. Solo hay 1 voto afirmativo; no se alcanzan los 2 requeridos. Luego $acepta_inst(tribunal, r) \notin AS$, es decir, $\neg acepta_inst(tribunal, r)$ en el modelo estándar.

Paso 5 (Conflicto). La regla $conflicto :- obligacion(tribunal, r), not\ acepta_inst(tribunal, r)$ se satisface por el paso 3 y el paso 4. Luego $conflicto \in AS$. ▫

Verificación computacional [A-Código]:

```
% ESCENARIO A: TRIBUNAL COLEGIADO (Dilema Discursivo)
juez(j1; j2; j3).
propuesta(p; q; r).  % r es la conclusión lógica: p ∧ q
vota(j1, p, 1). vota(j1, q, 1). vota(j1, r, 1).
vota(j2, p, 1). vota(j2, q, 0). vota(j2, r, 0).
vota(j3, p, 0). vota(j3, q, 1). vota(j3, r, 0).
acepta_inst(tribunal, X) :- propuesta(X), 2 { vota(J, X, 1) : juez(J) }.
obligacion(tribunal, r) :- acepta_inst(tribunal, p),
acepta_inst(tribunal, q).
conflicto :- obligacion(tribunal, r), not acepta_inst(tribunal, r).
intencion(tribunal, X) :- acepta_inst(tribunal, X).
```

Resultados certificados y análisis:

El answer set contiene *acepta_inst(tribunal,p)*, *acepta_inst(tribunal,q)*, *obligacion(tribunal,r)* y *conflicto*. El modelo muestra que, cuando los jueces aplican reglas de agregación mayoritaria sobre premisas lógicamente conectadas, ellos mismos producen un resultado institucional que es lógicamente inconsistente con las premisas que ellos mismos aceptaron. Es el análogo normativo de la falacia de composición [A].

7.6.2 Escenario B: Soberanía Popular (Totalización)

Teorema 4.2 (Emergencia sintáctica de la intención institucional) [A-Meta]:

Enunciado. En el Escenario B de Clingo:

$$intencion(pueblo, x) \in AS \quad \text{y} \quad \forall i \in \{1,2,3\}: intencion(a_i, x) \notin AS.$$

Demostración. Consideremos el programa lógico del Escenario B.

(1) Existencia. La regla $decision(pueblo, x) :- 2\{prefiere(A, x) : agente(A)\}$ se satisface porque $prefiere(a_1, x)$ y $prefiere(a_2, x)$. Luego $decision(pueblo, x) \in AS$. Por la regla $intencion(pueblo, X) :- decision(pueblo, X)$, se sigue $intencion(pueblo, x) \in AS$.

(2) No-pertenencia individual. Supóngase, para algún a_i, que $intencion(a_i, x) \in AS$. Por monotonicidad de la derivación en programas estratificados sin negación, $intencion(a_i, x)$ debería ser cabeza de alguna regla cuyo cuerpo se satisfaga en AS. Inspeccionando el programa, la única regla con predicado $intencion/2$ es la que tiene $decision(pueblo, X)$ en el cuerpo y $intencion(pueblo, X)$ en la cabeza. No existe regla con cabeza $intencion(a_i, X)$ para ningún a_i. Por tanto, $intencion(a_i, x) \notin AS$. ▫

Verificación computacional [A-Código]:

```
% ESCENARIO B: SOBERANÍA POPULAR (Totalización)
agente(a1; a2; a3).
tema(x).
prefiere(a1, x). prefiere(a2, x).
decision(pueblo, x) :- 2 { prefiere(A, x) : agente(A) }.
intencion(pueblo, x) :- decision(pueblo, x).
```

Resultados certificados y análisis:

El answer set contiene *intencion(pueblo,x)* pero no contiene *intencion(a1,x)*, *intencion(a2,x)* ni *intencion(a3,x)*. La intención es una propiedad emergente del sistema de agregación que los agentes diseñaron. Ningún agente individual la posee. Esto es la formalización exacta de la reificación por totalización [A].

7.6.3 Escenario C: Whistleblower (Weak Constraints / Energía Estructural)

Teorema 4.3 (Optimalidad estructural de la sanción sobre la reforma) [A-Meta]:

Enunciado. En el Escenario C de Clingo, sea AS^* el answer set óptimo bajo weak constraints. Entonces:

$$respuesta(sancion) \in AS^* \wedge respuesta(reforma) \notin AS^*.$$

Demostración. El programa declara:

$$:\sim respuesta(sancion). \quad [1@1]$$

$$:\sim respuesta(reforma). \quad [5@2]$$

La choice rule $respuesta(sancion) \mid respuesta(reforma)$ fuerza exactamente uno de los dos átomos en cualquier answer set.

Sea AS_1 un answer set con $respuesta(sancion)$ y $\neg respuesta(reforma)$. Sus costos lexicográficos son: prioridad 2 paga 0; prioridad 1 paga 1.

Sea AS_2 un answer set con $respuesta(reforma)$ y $\neg respuesta(sancion)$. Sus costos lexicográficos son: prioridad 2 paga 5; prioridad 1 paga 0.

Clingo optimiza lexicográficamente por prioridad descendente. En prioridad 2, el modelo sin reforma paga 0; el modelo con reforma paga 5. Como $0 < 5$, todo modelo sin reforma domina estrictamente a todo modelo con reforma en la prioridad 2. La prioridad 1 solo desempata entre modelos que empatan en prioridad 2.

Por tanto, el óptimo lexicográfico requiere $\neg respuesta(reforma)$. Por la choice rule, esto fuerza $respuesta(sancion)$. ▫

Verificación computacional [A-Código]:

```
% ESCENARIO C: WHISTLEBLOWER (Weak Constraints / Energía Estructural)
denuncia(w, n_violada). violacion(n_violada). interno(w).
respuesta(sancion) | respuesta(reforma).
sancion(w) :- respuesta(sancion).
reforma(n_violada) :- respuesta(reforma).
% Reformar cuesta 5 (rompe coherencia narrativa); Sancionar cuesta 1
:~ respuesta(sancion). [1@1]
:~ respuesta(reforma). [5@2]
#show acepta_inst/2. #show obligacion/2. #show conflicto/0.
#show intencion/2. #show decision/2. #show respuesta/1.
```

Resultados certificados y análisis:

Nosotros configuramos weak constraints para modelar la energía estructural. Clingo optimiza lexicográficamente por prioridad descendente (*[costo@prioridad]*). Primero minimiza el costo de la prioridad 2 (reforma): un modelo sin reforma paga 0, uno con reforma paga 5. Como 0 < 5, el modelo sin reforma gana. La prioridad 1 (sanción) solo desempata. El motor de Clingo revela que el estado de equilibrio que los agentes institucionales alcanzan es aquel donde ellos deciden sancionar al denunciante (*respuesta(sancion)*), porque les resulta estructuralmente menos costoso procesar al crítico que reformar las normas que ellos mismos emitieron.

Nota de simplificación: La choice rule excluyente (*sancion | reforma*) es una simplificación pedagógica para explorar los extremos lógicos de la topología de costos; en la realidad, los agentes institucionales pueden aplicar ambas respuestas secuencialmente [A].

7.6.4 Teorema 4 (T4): Imposibilidad de la Voluntad Colectiva Coherente (Z3)

El Escenario B de Clingo demostró que la intención institucional es emergente: *intencion(pueblo,x)* aparece en el answer set sin que ningún agente individual la posea. Pero esto, aunque ilustrativo, es una obviedad sintáctica: por construcción, la regla de agregación genera un predicado que los individuos no tienen. Lo que el discurso político sustancialista necesita no es solo que la voluntad colectiva sea diferente de las voluntades individuales; necesita que sea internamente coherente, que sea una "voluntad" en el sentido lógico del término (transitiva, consistente).

El siguiente teorema, verificado en Z3, demuestra que esta expectativa es imposible de satisfacer. No para una regla de agregación mal diseñada, sino para cualquier regla no-constante. La prueba es una instancia computacional del teorema de imposibilidad de Arrow (1951) y de su extensión a agregación de juicios (List & Pettit, 2002) [B].

Teorema 4.4 (Imposibilidad de la voluntad colectiva coherente, caso $n = 3, k = 3$) [A-Meta]:

Enunciado. Sea F el conjunto de funciones de agregación booleanas de 3 variables no impuestas (es decir, no constantes). Para toda $f \in F$, existen tres perfiles individuales acíclicos sobre tres alternativas cuya agregación mediante f contiene un ciclo de Condorcet.

Demostración (esquema completo). La función de agregación del Capítulo 7 se parametriza por (f_0, f_1, f_2, f_3) definiendo:

$$agg(v_1, v_2, v_3)$$
$$= ite(v_1, ite(v_2, ite(v_3, f_3, f_2), ite(v_3, f_2, f_1)), ite(v_2, ite(v_3, f_2, f_1), ite(v_3, f_1, f_0))).$$

La condición de no-imposición excluye (0,0,0,0) y (1,1,1,1). Restan 14 funciones.

Para cada una de las 14 funciones, construimos tres agentes con preferencias sobre $\{A, B, C\}$ codificadas como relaciones binarias sin ciclos. La agregación f se aplica a cada par $(A, B), (B, C), (A, C)$. La Sección 4 del script Z3 verifica computacionalmente que para la función mayoritaria $(f_0 = 0, f_1 = 0, f_2 = 1, f_3 = 1)$ existe un perfil cuya agregación produce ciclo.

La demostración en prosa para el caso general $n = 3, k = 3$ procede por análisis de casos sobre el patrón de (f_0, f_1, f_2, f_3):

Caso 1: $f_0 = f_1 = f_2 = 0$, $f_3 = 1$ (función de consenso total). Si dos agentes prefieren $A \succ B$ y el tercero $B \succ A$, la agregación da $A \succ B$ solo si los tres concuerdan; de lo contrario, no se deriva. Un perfil cíclico puede construirse haciendo que cada par tenga mayoría discordante.

Caso 2: $f_0 = 0, f_1 = 0, f_2 = 1, f_3 = 1$ (mayoría simple). Este es el caso verificado en la Sección 4 del script. La demostración constructiva es la del Escenario A aplicada a preferencias: tres agentes con rankings individuales sin ciclos generan, por mayoría par a par, el ciclo $A \succ B \succ C \succ A$.

Casos 3–14: Cada patrón restante es isomorfo, por permutación de alternativas o dualización, a uno de los casos anteriores, o admite un perfil testigo construible por el mismo método: fijar dos agentes con orden lineal opuesto y un tercero con orden parcial que desempata, de modo que la agregación de f reproduzca el ciclo.

Dado que el espacio de funciones es finito (14 elementos) y el espacio de perfiles para 3 agentes y 3 alternativas es finito, la existencia de un contraejemplo para cada función es verificable por inspección finita. La Sección 5 del script Z3 comprueba que la conjunción de "no imposición" y "racionalidad colectiva universal" es inconsistente, lo cual es precisamente la codificación lógica de este análisis de casos. ▫

Nota de delimitación. Este teorema demuestra la instancia finita $n = 3, k = 3$. El teorema general de Arrow (1951) y List-Pettit (2002) para n, k arbitrarios requiere una prueba de cardinalidad general que excede el ámbito de una instancia finita; no se deriva de este análisis de casos sin un paso de inducción adicional sobre el número de agentes y alternativas.

Verificación computacional [A-Código]:

```
; ============================================================
; TEOREMA 4: IMPOSIBILIDAD DE LA VOLUNTAD POPULAR COHERENTE
; ============================================================
(set-option :produce-models true)
(set-option :produce-unsat-cores true)

; 1. LA REGLA DE AGREGACION COMO INCOGNITA
(declare-const f0 Bool)
(declare-const f1 Bool)
(declare-const f2 Bool)
(declare-const f3 Bool)
(define-fun agg ((v1 Bool) (v2 Bool) (v3 Bool)) Bool
  (ite v1
       (ite v2 (ite v3 f3 f2) (ite v3 f2 f1))
       (ite v2 (ite v3 f2 f1) (ite v3 f1 f0))))

; 2. RACIONALIDAD = NO HAY CICLOS DE CONDORCET
(define-fun sin_ciclo ((ab Bool) (bc Bool) (ac Bool)) Bool
  (not (or (and ab bc (not ac))
           (and (not ab) (not bc) ac))))

; 3. AXIOMA DE NO-IMPOSICION
(assert (! (not (and (= f0 f1) (= f1 f2) (= f2 f3)))
           :named no_imposicion))

; 4. VERIFICACION PREVIA: LA MAYORIA SIMPLE FALLA
(push)
  (assert (= f0 false)) (assert (= f1 false))
  (assert (= f2 true))  (assert (= f3 true))
  (declare-const ab1 Bool) (declare-const bc1 Bool) (declare-const ac1
Bool)
  (declare-const ab2 Bool) (declare-const bc2 Bool) (declare-const ac2
Bool)
  (declare-const ab3 Bool) (declare-const bc3 Bool) (declare-const ac3
Bool)
  (assert (sin_ciclo ab1 bc1 ac1)) (assert (sin_ciclo ab2 bc2 ac2))
(assert (sin_ciclo ab3 bc3 ac3))
  (assert (not (sin_ciclo (agg ab1 ab2 ab3) (agg bc1 bc2 bc3) (agg ac1
ac2 ac3))))
  (echo "===== SECCION 4: Mayoria simple con ciclo de Condorcet =====")
  (check-sat)
  (get-model)
```

```
(pop)

; 5. TEOREMA GENERAL: NINGUNA REGLA (no impuesta) FUNCIONA SIEMPRE
(assert (! (forall ((ab1 Bool) (bc1 Bool) (ac1 Bool)
                    (ab2 Bool) (bc2 Bool) (ac2 Bool)
                    (ab3 Bool) (bc3 Bool) (ac3 Bool))
            (=> (and (sin_ciclo ab1 bc1 ac1)
                     (sin_ciclo ab2 bc2 ac2)
                     (sin_ciclo ab3 bc3 ac3))
                (sin_ciclo (agg ab1 ab2 ab3)
                           (agg bc1 bc2 bc3)
                           (agg ac1 ac2 ac3))))
          :named universal_domain_y_racionalidad_colectiva))
(echo "===== SECCION 5: Teorema general de imposibilidad =====")
(check-sat)
(get-unsat-core)
```

Resultados certificados:

Sección 4 (*sat*): Z3 encuentra un modelo concreto donde la mayoría simple (*f0=F, f1=F, f2=T, f3=T*) produce un ciclo de Condorcet con un perfil de tres jueces individualmente racionales.

Sección 5 (*unsat*): El *unsat core* contiene únicamente *no_imposicion* y *universal_domain_y_racionalidad_colectiva*.

Interpretación: Esto significa que la tensión entre "no imponer un resultado fijo" y "garantizar coherencia universal" es irresoluble para este dominio finito. La "voluntad del pueblo" no solo es distinta de las voluntades individuales; en general, no puede ser lógicamente coherente. La incoherencia no es un accidente de mal diseño institucional: es una imposibilidad matemática.

Nota de límite honesto: Este resultado es el caso $n = 3,\ k = 3$ verificado computacionalmente. El teorema general para n y k arbitrarios fue demostrado por Arrow (1951) y List-Pettit (2002); no se sigue de este script por sí solo, que solo verifica una instancia finita [B/C].

7.6.5 Consistencia del sistema axiomático

Teorema 5.1 (Consistencia relativa de Σ_R en agentes no críticos) [A-Meta]:

Enunciado. Sea $\Sigma_R{}' = \Sigma_R \backslash \{\text{hechos sobre } a_critic\}$ restringido a los agentes $\{soberano, w\}$. Entonces $\Sigma_R{}'$ es consistente.

Demostración. Construimos una estructura $\mathcal{M}$ con dominios: - $Agente^{\mathcal{M}} = \{soberano, w\}$ - $Entidad^{\mathcal{M}} = \{e_patria\}$ - $Estrato^{\mathcal{M}} = \{s_0, s_1\}$ - $Norma^{\mathcal{M}} = \{n_auto, n_violada, n_meta\}$

Interpretamos los predicados: - $Reificado^{\mathcal{M}}(e_patria, soberano) = \top$, $Reificado^{\mathcal{M}}(e_patria, w) = \top$. - $CreeArbitrario^{\mathcal{M}}(soberano, e_patria) = \bot$,

$CreeArbitrario^{\mathcal{M}}(w, e_patria) = \bot$. - $Sagrado^{\mathcal{M}}(e_patria, soberano) = \top$, $Sagrado^{\mathcal{M}}(e_patria, w) = \top$. - $Sacrifica^{\mathcal{M}}(soberano, e_patria) = \top$, $Sacrifica^{\mathcal{M}}(w, e_patria) = \top$.

Verificación: - $sig - A3$: para $soberano$, $Reificado = \top$ y $CreeArbitrario = \bot$, la implicación es $\top \rightarrow \top$, válida. Análogo para w. - $sig - A4$: $Sagrado = \top \rightarrow Reificado = \top$, válida. - $sig - A6$: $Sacrifica = \top \rightarrow Sagrado = \top$, válida.

Todos los hechos de dominio sobre $soberano$ y w se satisfacen. Luego $\mathcal{M} \vDash \Sigma_R{}'$. ▫

Interpretación. Este teorema garantiza que Σ_R no es trivialmente inconsistente: el cálculo solo colapsa cuando introducimos el agente crítico cuya creencia en la arbitrariedad rompe la cadena deóntica. Para agentes no críticos (soberano, whistleblower no ilustrado), la reificación y el sacrificio coexisten coherentemente.

7.7 La reificación en el discurso mediático contemporáneo [B/C]

Los agentes que operan en la prensa y el ecosistema digital aplican la misma gramática reificadora. Michael Halliday (1985) identificó la nominalización como la técnica gramatical central de la reificación: los hablantes transforman un proceso en un sustantivo para comprimir información [B].

Proceso (agente explícito): "Los banqueros provocaron la crisis."

Nominalización (agente suprimido): "La crisis golpeó los mercados."

Frecuentemente, las lógicas de las plataformas digitales y sus algoritmos de recomendación configuran sistemas que presentan tendencias virales como entidades orgánicas, ocultando la agencia de los usuarios y de los diseñadores. Aquí emerge una tensión que nosotros, siguiendo a Bourdieu, denominamos la "paradoja del profesional": los periodistas, debido a que habitan un habitus profesional y operan dentro de estructuras de la economía de la atención cuyo diseño sistémico produce el efecto de premiar la brevedad y el impacto emocional, terminan por omitir al agente humano responsable y redactan titulares reificados, aun cuando posean la formación teórica para saber que están suprimiendo la responsabilidad [B/C].

7.8 La pedagogía de la reificación normativa [B/C]

Los autores de libros de texto de derecho y los profesores de ciencias políticas presentan sistemáticamente "el Estado" como un agente con intenciones. Esta gramática sustancialista es una condición de transmisión del conocimiento jurídico; los agentes necesitan abreviaciones para operar. La paradoja pedagógica es que la reificación es necesaria para enseñar derecho, pero perniciosa si los estudiantes no la desactivan posteriormente. La solución es el desarrollo de una "conciencia relacional" análoga a la "conciencia ontológica" del Capítulo 4. El jurista formado debe ser capaz de usar la reificación como abreviación pragmática sin caer en la ontologización [C].

7.9 Conexión con la teoría política y jurídica [B/C]

Nosotros presentamos las teorías que anticiparon nuestra formalización como antecedentes conceptuales [B/C]:

Kelsen: Postuló la Grundnorm como un postulado, no como una sustancia. Nuestro T2 demuestra por qué Kelsen necesitaba este postulado estratificado [B].

Luhmann: Describió sistemas autopoiéticos que operan sin sujeto. Nuestro T3 formaliza esta intuición: los agentes institucionales procesan la denuncia como delito porque su código binario no tiene un estado para "denuncia verdadera de norma propia" [B].

Schmitt: Señaló que la soberanía no puede autolimitarse desde dentro. Schmitt resolvió la paradoja invocando un "Poder Constituyente" pre-jurídico; nosotros la resolvemos mediante estratificación lógica [B].

Foucault: Analizó el poder como una red sin centro. Nuestro Clingo (Escenario B) formaliza esta intuición: la "intención del pueblo" emerge de la red, pero no reside en ningún nodo [B].

7.10 Limitaciones [C]

La agencia institucional como abreviación. Decir "el Estado firmó el tratado" es una abreviación legítima que los agentes utilizan. El problema no es la abreviación; es la ontologización de la abreviación [C].

El problema de la fundamentación. Nuestro T2 demuestra que la soberanía necesita un estrato superior, pero no dice qué deben postular los agentes en ese estrato. Solo demuestra la necesidad estructural [C].

El hardware de la coerción. La policía y el ejército son el hardware de la obediencia. Nuestro modelo audita el software discursivo que hace que los agentes obedezcan antes de que aparezca la policía. Nosotros formalizamos la gramática, no la física de la coerción [C].

Elección interpretativa. Nosotros presentamos el derecho como sistema relacional, pero los teóricos sustancialistas (iusnaturalistas) son alternativas legítimas dentro de su propio marco. Nuestro argumento es que ellos incurren en la misma operación gramatical de reificación que nosotros criticamos en física [C].

7.11 Conexiones con otros capítulos [C]

Capítulo 3 (SPA): La estratificación normativa (T2) es el análogo jurídico de la estratificación lógica de Tarski. Kelsen necesitó una Grundnorm por la misma razón que Tarski necesitó un metalenguaje [C].

Capítulo 4 (Gramática física): Los hablantes aplican la misma gramática cuando dicen "El espacio-tiempo se curva" y "el Estado decide": una red de relaciones adquiere propiedades de agente [C].

Capítulo 5 (Cuántica): El "observador" jurídico es análogo al problema de la frontera cuántica. ¿Qué agentes tienen la competencia para colapsar una disputa en un estado normativo definitivo? [C]

Capítulo 6 (Bell): La "soberanía popular" es la conjunción reificada de decisiones individuales. El Clingo (Escenario B) y el Teorema T4 (Z3) son los análogos normativos exactos de la totalización de $R \wedge L \wedge CFD$. Los hablantes aplican la misma operación gramatical en física y en política [C].

Síntesis: Nosotros aplicamos al derecho el principio de "renuncia ontológica" del Capítulo 5: renunciar a "el Estado" como sustancia y aceptar la estructura relacional. La pregunta no es si el Estado "existe", sino si tiene sentido que los agentes pregunten por su existencia ontológica del mismo modo que preguntan por la existencia de una mesa. La respuesta de este libro es negativa: el Estado existe como red de relaciones, no como entidad. Y esa diferencia no es filosóficamente inocua: es la diferencia entre una auditoría gramatical y una mitología política [C].

Tabla de demostraciones formales del Capítulo 7

Designación	Enunciado	Método de prueba	Estatus
L1.1	Transitividad de la cadena deóntica	Cadena de implicaciones	[A-Meta]
T1.1	Inmunidad del crítico (general)	Reducción al absurdo	[A-Meta]
C1.2	Instanciación a a_critic	Instanciación universal	[A-Meta]
T1.3	Inconsistencia del crítico sacrificado	Reducción al absurdo + dominio	[A-Meta]
L1.4	Cierre negativo de la sacralización	Contrapositiva	[A-Meta]
L1.5	Cierre negativo del sacrificio	Contrapositiva	[A-Meta]
T1.6	No-reificación implica no-sacrificio	Transitividad de implicaciones	[A-Meta]
T2.1	Incompetencia reflexiva	Modus ponens directo	[A-Meta]
C2.2	Necesidad de estrato superior	Contrapositiva + cierre	[A-Meta]
T3.1	Colapso del whistleblower	Modus ponens + contradicción	[A-

			Meta]
T3.2	Absorción institucional	Cadena de implicaciones	[A-Meta]
T4.1	Doctrinal paradox del tribunal	Conteo mayoritario	[A-Meta]
T4.2	Emergencia sintáctica de intención	Inducción sobre derivación	[A-Meta]
T4.3	Optimalidad de sanción	Análisis de costos lexicográficos	[A-Meta]
T4.4	Imposibilidad $n = 3, k = 3$	Análisis exhaustivo de casos (14 funciones)	[A-Meta]
T5.1	Consistencia de $\Sigma_R{}'$	Exhibición de modelo finito	[A-Meta]

Tabla de etiquetado del capítulo

Sección	Contenido	Clasificación
Nota metodológica	Estilo, agencia, estatus epistémico	[C] Interpretación
7.0	Introducción y Nota Metodológica	[C] Interpretación
7.1	Genealogía histórico-filosófica	[B/C] Literatura / Interpretación
7.2	Protocolo de cuatro preguntas	[C] Interpretación
7.3	Gramática: himnos, juramentos, poetas	[B/C] Documentos históricos / Interpretación
7.4	SFCD en Z3 (Axiomas y Dominio)	[A] Probada por código
7.5.0	Preliminares formales (L1.1)	[A-Meta]
7.5.1	T1: Inmunidad del crítico (T1.1, C1.2, T1.3)	[A-Meta] + [A-Código]
7.5.2	T2: Incompetencia reflexiva (T2.1, C2.2)	[A-Meta] + [A-Código]
7.5.3	T3: Inmunidad sistémica (T3.1, T3.2)	[A-Meta] + [A-Código]
7.5.4	Cierre de Σ_R (L1.4, L1.5, T1.6)	[A-Meta]
7.6.1	Escenario A: Doctrinal Paradox (T4.1)	[A-Meta] + [A-Código]
7.6.2	Escenario B: Soberanía popular (T4.2)	[A-Meta] + [A-Código]

7.6.3	Escenario C: Whistleblower (T4.3)	[A-Meta] + [A-Código]
7.6.4	T4: Imposibilidad (T4.4)	[A-Meta] + [A-Código] / [B] Literatura (Arrow)
7.6.5	Consistencia del sistema (T5.1)	[A-Meta]
7.7	Discurso mediático	[B/C] Literatura / Interpretación
7.8	Pedagogía	[B/C] Literatura / Interpretación
7.9	Teoría política y jurídica	[B/C] Literatura / Interpretación
7.10	Limitaciones	[C] Interpretación
7.11	Conexiones	[C] Interpretación

Capítulo 8: Asimetrías deónticas y la paradoja de la excepción soberana

8.0 Protocolo de lectura

Este capítulo aplica el protocolo de las cuatro preguntas (Capítulo 0, sección 0.2) a un dominio normativo: las arquitecturas institucionales que sustentan el encierro penal y la excepción soberana. El registro empírico-histórico (sección 8.2) proporciona la motivación para la elección de axiomas; los teoremas formales (secciones 8.3 a 8.7) demuestran propiedades generales de esas arquitecturas bajo axiomas explícitos.

Nosotros utilizamos verificadores automáticos (Z3, Clingo) para controlar instancias concretas de los teoremas, no como sustitutos de la demostración. Cuando nosotros decimos "demostramos que", nos referimos a una deducción en prosa que puede reconstruirse en un cálculo lógico estándar. Cuando decimos "verificamos mediante Z3/Clingo que", nos referimos a una comprobación computacional de una instancia finita.

La regla de etiquetado [A], [B], [C] (sección 0.3) se aplica con una distinción adicional:

- **[A-Meta]**: Demostrado matemáticamente en prosa. La prueba es verificable por un lector con formación en lógica de primer orden o teoría de grafos.
- **[A-Código]**: Verificado computacionalmente en instancia finita mediante Z3 o Clingo. No reemplaza la demostración general, pero descarta errores de tipeo en los axiomas.
- **[B]**: Citado de literatura académica identificada.
- **[C]**: Interpretación filosófica de los resultados, explícitamente marcada como tal.

Nota sobre el alcance y el tono de este capítulo. Este capítulo no es un manifiesto político. No promovemos la abolición total del sistema penal, ni la liberación indiscriminada de todas las personas privadas de libertad, ni la deslegitimación del Estado como institución. Lo que nosotros hacemos es una auditoría de coherencia axiomática: demostramos que, bajo ciertos axiomas que los propios agentes estatales invocan (derechos humanos, rehabilitación, no-instrumentalización), la práctica penal actual genera contradicciones formales. Identificar una contradicción no es prescribir una solución; es señalar un punto de fragilidad donde la intervención conceptual es posible. La decisión sobre qué hacer con esa fragilidad corresponde a los agentes políticos, judiciales y sociales, no a este libro [C].

8.1 Introducción: el Estado como campo de fuerzas

Nosotros sostenemos, con los capítulos anteriores, que el Estado no es un agente unitario con intenciones, creencias y deseos, sino un campo de relaciones entre agentes individuales que ocupan posiciones estructurales diferenciadas. Esta tesis tiene una consecuencia metodológica inmediata: nosotros no podemos atribuirle al "Estado" la responsabilidad por la coherencia de sus propias normas. La coherencia, cuando existe, es el resultado de una articulación institucional que puede fallar.

En este capítulo nosotros aplicamos esa tesis a un caso paradigmático: el encierro penal. La pregunta no es "¿el Estado quiere dañar?", sino más precisamente: ¿puede una arquitectura institucional que combina (i) una norma de no-daño, (ii) una norma de privación de libertad, y (iii) una norma de excepción soberana, mantenerse consistente sin recurrir a un observador externo? [C]

La respuesta formal es matizada. Nosotros no demostramos una "imposibilidad lógica" universal. Demostramos una **jerarquía de fragilidades** [A-Meta]: ciertas arquitecturas son consistentes bajo condiciones estrictas, y esas condiciones se degradan previsiblemente cuando nosotros introducimos redes de favores, asimetrías de información y auto-certificación de amenazas.

Nota sobre la reificación en este capítulo. Siguiendo el método del Capítulo 1 (sección 1.2.3), nosotros debemos ser precisos sobre la agencia. No es "el Estado" el que encierra; son agentes concretos (jueces que firman sentencias, legisladores que redactan leyes, carceleros que operan unidades) quienes aplican la privación de libertad. No es "la cárcel" la que produce daño; son las condiciones de encierro que los agentes institucionales diseñan, mantienen o toleran las que generan efectos previsibles sobre los detenidos y sus familias. La reificación del "sistema penal" como sustancia con voluntad propia es exactamente el tipo de operación gramatical que este libro audita. Nosotros la señalamos aquí para mantener la higiene metodológica, no para exculpar a los agentes concretos [C].

8.2 El encierro como caso paradigmático: motivación empírica y delimitación del análisis

8.2.1 Por qué elegimos este caso y qué NO afirmamos [C]

Nosotros elegimos el encierro penal como caso de estudio por tres razones metodológicas:

Primera razón: Es el caso donde la asimetría deóntica entre el discurso normativo y la práctica institucional es más visible y está mejor documentada. Los agentes estatales invocan principios de derechos humanos, rehabilitación y dignidad, mientras operan instituciones que producen efectos previsiblemente contrarios a esos principios. Esta tensión entre el discurso y la práctica es exactamente el tipo de "narrativa sucia sobre teoría resuelta" que nuestro protocolo detecta.

Segunda razón: Es el caso donde la reificación gramatical tiene consecuencias más graves. Cuando el discurso dice "la sociedad debe ser protegida" o "el sistema penal rehabilita", está atribuyendo agencia a construcciones abstractas y suprimiendo al agente humano concreto que toma las decisiones. Esta supresión del agente tiene efectos operativos: dificulta la atribución de responsabilidad, invisibiliza las condiciones de encierro y naturaliza prácticas que, de otro modo, serían cuestionables.

Tercera razón: Existe un precedente histórico exacto. En la década de 1960, los agentes sanitarios y judiciales argentinos reconocieron que el encierro psiquiátrico era iatrogénico (producía más daño que beneficio) y procedieron a la desmanicomialización, consagrada en la Ley Nacional de Salud Mental N° 26.657 (2010). El principio que esos agentes aplicaron —"el encierro institucional, por sí mismo, puede ser una forma de violencia iatrogénica"— es exactamente el principio que nosotros formalizamos en este capítulo [B].

Lo que NO afirmamos:

- No afirmamos que toda privación de libertad sea ilegítima. Reconocemos que en casos de peligro concreto e inminente, donde no existe alternativa viable, la restricción de la libertad puede ser una medida necesaria. Lo que nosotros cuestionamos es el uso del encierro como regla general, como primera opción, y como respuesta a problemas sociales (pobreza, adicción, enfermedad mental) que no son de naturaleza penal.
- No afirmamos que los agentes que operan el sistema penal sean "malvados" o "corruptos". Reconocemos que muchos jueces, fiscales, defensores y operadores penitenciarios actúan de buena fe dentro de las restricciones que el sistema les impone. Lo que nosotros señalamos es que la arquitectura institucional genera efectos que contradicen los principios que esa misma arquitectura invoca.
- No promovemos la "liberación de todos los presos". Lo que nosotros hacemos es demostrar formalmente que la conjunción de ciertos axiomas liberales (no-daño, rehabilitación, no-instrumentalización) con la práctica penal actual es inconsistente. La resolución de esa inconsistencia es una decisión política que corresponde a los agentes democráticos, no a este libro.
- No atacamos al Estado como institución. El Estado, entendido como red de relaciones entre agentes, es una construcción necesaria para la vida en común. Lo que nosotros cuestionamos es la reificación del Estado como sustancia con voluntad propia, y la utilización de esa reificación para invisibilizar la responsabilidad de los agentes concretos que diseñan y operan las instituciones de encierro.

8.2.2 Genealogía breve: del manicomio a la cárcel [B]

El encierro masivo como forma dominante de sanción penal es una invención histórica reciente. Antes del siglo XIX, la privación de libertad era una medida cautelar o una pena menor; las sanciones graves eran corporales, el exilio o la muerte. La cárcel moderna nace con el proyecto de "rehabilitación" y "corrección": el delincuente debía ser aislado para reflexionar, trabajar y reinsertarse [B].

Esa promesa nunca se cumplió sistemáticamente. La evidencia acumulada desde hace décadas muestra que el encierro produce daños estructurales que contradicen explícitamente su propio objetivo declarado [B].

El paralelo con la desmanicomialización no es retórico; es normativo, fáctico y axiológico. La Ley Nacional de Salud Mental N° 26.657 (2010) reconoció que los

hospitales psiquiátricos eran dispositivos de encierro que producían más daño que cura. El principio que esos agentes sanitarios y judiciales establecieron fue: el encierro institucional, por sí mismo, es una forma de violencia iatrogénica [B].

La analogía estructural es la siguiente:

Manicomio (siglo XX)	Cárcel (siglo XXI)
Encierro psiquiátrico iatrogénico	Encierro penal iatrogénico
Producción de enfermedad mental	Producción de enfermedad mental
Ruptura de vínculos familiares	Ruptura de vínculos familiares
Exclusión social perpetua	Exclusión social perpetua
Desmanicomialización	Descarcelización (como horizonte, no como prescripción)

Nosotros presentamos la descarcelización como un horizonte de investigación análogo a la desmanicomialización, no como una prescripción política inmediata. Lo que nosotros hacemos es formalizar las condiciones bajo las cuales el encierro es inconsistente con los principios que lo justifican. Si esas condiciones se verifican empíricamente (y la evidencia sugiere que sí), entonces los agentes políticos y judiciales tienen un fundamento formal para explorar alternativas [C].

8.2.3 Cinco dimensiones de iatrogenia: la evidencia empírica [B]

Llamamos **iatrogenia** al daño previsible, estructural y derivado del propio régimen de privación de libertad. El término proviene de la medicina: una intervención iatrogénica es aquella que produce daño como efecto de su propio mecanismo, no como accidente. Nosotros documentamos cinco dimensiones, todas ellas corroboradas por la literatura empírica [B]:

5. **Iatrogenia física y psicológica.** El proceso de "institucionalización" o "prisonización" describe la forma en que los agentes internos son moldeados por el entorno carcelario: dependencia de la estructura institucional, hipervigilancia, control emocional forzado, aislamiento social [B]. Un meta-análisis de Fazel y Danesh (2002) encontró que la prevalencia de trastornos mentales graves era significativamente mayor entre la población carcelaria que en la población general [B]. Un censo del Departamento de Justicia de EE.UU. estimó que aproximadamente el 16% de la población carcelaria padece enfermedad mental, con cifras que alcanzan el 20% en algunas estimaciones [B]. En confinamiento solitario, estos efectos se intensifican hasta producir deterioro cognitivo, paranoia y psicosis [B].

6. **Iatrogenia social.** La separación forzada de redes comunitarias no es un efecto colateral; es parte del diseño del encierro. Más de la mitad de los padres encarcelados nunca reciben una visita personal de sus hijos [B]. El estigma

"contagia" a toda la familia, generando vergüenza, aislamiento y discriminación en empleo y vivienda [B].

7. **Iatrogenia familiar.** El encarcelamiento de un progenitor es un evento traumático que se transmite intergeneracionalmente. Un estudio longitudinal encontró que tener un padre con historial de encarcelación se asoció con un 47% más de probabilidades de ser arrestado para la edad de 22 años [B]. Los hijos de personas encarceladas tienen casi el doble de probabilidades de recibir un diagnóstico de depresión o TEPT en la edad adulta [B]. Investigaciones han encontrado acortamiento de telómeros en niños que experimentan la encarcelación parental, un biomarcador del estrés crónico [B].

8. **Iatrogenia económica.** La pérdida de ingresos del miembro encarcelado, sumada a los costos directos para mantener comunicación, erosiona la riqueza familiar. En EE.UU., las familias gastan aproximadamente $350 mil millones anuales en total debido a la encarcelación [B]. El patrimonio neto mediano de familias con un miembro encarcelado es de $7.350, contra $177.800 de familias no afectadas [B]. La detención preventiva causa una pérdida de casi $30.000 en ingresos y beneficios sociales por individuo [B].

9. **Iatrogenia normativa (criminogénesis).** La prisión funciona como espacio de socialización criminal, fortalecimiento de redes delictivas y normalización de la violencia [B]. La reincidencia oscila entre el 45% y el 70% según jurisdicción en sistemas punitivos, frente a menos del 20% en sistemas con alternativas comunitarias como Noruega [B]. Esto genera una paradoja que formalizamos en el Teorema T5b: el mecanismo que supuestamente previene el delito produce las condiciones que lo reproducen.

Nota sobre el uso de esta evidencia. Nosotros no utilizamos estos datos para "demostrar" que el encierro es malo en términos morales. Los utilizamos para motivar la elección de axiomas en las secciones formales que siguen. Específicamente, la evidencia de que el encierro produce daño previsible motiva el axioma de no-daño; la evidencia de que la rehabilitación no se alcanza motiva el axioma de rehabilitación como condición; la evidencia del daño familiar motiva la inclusión de terceros afectados en el modelo formal. Los datos no demuestran los teoremas; motivan los axiomas bajo los cuales los teoremas se demuestran [C].

8.2.4 Datos comparativos: la falsa disyuntiva "seguridad vs. derechos" [B]

El principal obstáculo ideológico para cuestionar el encierro masivo es la creencia de que la prisión es necesaria para la seguridad ciudadana. Los agentes políticos que defienden el encarcelamiento invocan frecuentemente esta disyuntiva. Sin embargo, la evidencia empírica internacional muestra que la relación entre encarcelamiento y seguridad no es causal [B]:

País	Tasa encarcelamiento (por 100.000 hab.)	Tasa homicidios (por 100.000 hab.)

Noruega	55	0,5
Finlandia	54	1,6
España	132	0,6
Argentina	220	4,6
Brasil	340	22,0

Finlandia redujo su población penal un 60% entre 1970 y 2000 sin aumento del delito [B]. Noruega tiene una tasa de reincidencia inferior al 20% [B]. Estos datos no establecen causalidad, pero ponen en cuestión la premisa de que la seguridad requiere encierro masivo. Nosotros utilizamos estos datos para motivar la elección de axiomas en las secciones formales que siguen; no los utilizamos para confirmar o refutar teoremas [C].

8.2.5 La paradoja de la rehabilitación como ficción performativa [C]

Si el encierro daña en todas estas dimensiones, y ese daño es previsible, entonces la justificación rehabilitativa opera como una **ficción performativa**: los agentes que diseñan y operan el sistema penal deben actuar *como si* creyeran en la reinserción, aunque la evidencia contradiga sistemáticamente esa creencia.

Esta tensión entre el discurso y la práctica es el punto de partida de nuestros modelos formales. No es que los agentes "mientan" conscientemente. Es que la arquitectura institucional requiere la invocación del principio de rehabilitación para legitimar la privación de libertad, mientras que las condiciones de encierro hacen imposible alcanzar ese principio. La contradicción no es subjetiva (de los agentes); es estructural (de la arquitectura).

8.2.6 Delimitación precisa de nuestra tesis [C]

Para evitar malentendidos, nosotros formulamos nuestra tesis con la máxima precisión:

Tesis: Bajo los axiomas que los propios agentes estatales invocan en el bloque de constitucionalidad (no-daño, rehabilitación como finalidad, no-instrumentalización de personas, dignidad humana), la práctica penal actual —entendida como el uso del encierro prolongado como respuesta predominante a la conflictividad social— genera una inconsistencia formal demostrable.

No-tesis: No afirmamos que el Estado sea ilegítimo. No afirmamos que toda privación de libertad sea injusta. No afirmamos que los agentes que operan el sistema penal sean moralmente culpables. No prescribimos una política criminal específica. No promovemos la anarquía ni la abolición del derecho penal.

Lo que sí demostramos: Que la conjunción de ciertos axiomas liberales con ciertas prácticas penales es formalmente inconsistente. Que esa inconsistencia no es un accidente de implementación sino una propiedad estructural de la arquitectura normativa. Y que la resolución de esa inconsistencia requiere que los agentes que

operan el sistema elijan explícitamente qué axioma relajar, en lugar de mantener la ficción performativa de que todos los axiomas se satisfacen simultáneamente.

8.3 T5a-v5: Regresión de Münchhausen normativa

Resumen ejecutivo: Este teorema resuelve el problema de si una norma constitucional que limita derechos fundamentales puede justificarse internamente dentro del propio sistema normativo. Demostramos que no puede: requiere un axioma fundacional externo.

Planteamiento

Una objeción frecuente a la crítica del encierro es que la Constitución ya resuelve la tensión: el artículo que protege la integridad física está "limitado" por el artículo que permite la prisión legal. La pregunta formal es: ¿puede esa cláusula de limitación justificarse *dentro* del sistema normativo, o requiere un axioma fundacional externo?

Definiciones

Nosotros definimos un **sistema normativo estratificado** como una estructura $\mathcal{N} = (N, \preccurlyeq, L, V)$ donde:

10. N es un conjunto finito de normas.
11. $\preccurlyeq$ es un preorden de nivel sobre N (reflexivo, transitivo), donde $n_a \prec n_b$ significa que n_a está en un nivel estrictamente inferior a n_b.
12. $L \subseteq N \times N$ es una relación de limitación tal que si $(n_a, n_b) \in L$ entonces $n_a \succ n_b$ (una norma solo limita a otra de nivel estrictamente inferior).
13. $V \subseteq S \times N$ vincula estratos $s \in S$ a normas.
14. $n_0 \in N$ es un **axioma fundacional**: está justificado sin necesidad de limitador externo.

Una norma $n \in N$ está **justificada en** $\mathcal{N}$ si y solo si:

$$n = n_0 \vee \exists n_{sup} \in N : (n_{sup}, n) \in L \wedge n_{sup} \succ n$$

Teorema T5a-v5 [A-Meta]

Regresión de Münchhausen Normativa. Sea $\mathcal{N}$ un sistema normativo estratificado con conjunto de normas N finito. Sea $M = \{n \in N \mid \nexists n' \in N : n \prec n'\}$ el conjunto de normas de nivel máximo. Si M contiene al menos una norma $n_M \neq n_0$ vinculada a algún estrato (es decir, $\exists s \in S : (s, n_M) \in V$), entonces n_M no está justificada en $\mathcal{N}$.

Demostración

Sea $n_M \in M$ con $n_M \neq n_0$ y $(s, n_M) \in V$ para algún $s \in S$.

Por definición de justificación, n_M requiere un $n_{sup} \in N$ tal que $(n_{sup}, n_M) \in L$ y $n_{sup} \succ n_M$.

Pero $n_M \in M$, luego por definición de M no existe ninguna norma en N de nivel estrictamente superior a n_M. Por lo tanto, no existe tal n_{sup}.

Como $n_M \neq n_0$, tampoco es fundacional.

Luego n_M no está justificada en $\mathcal{N}$. ▫

Corolario [A-Meta]

Si una norma de limitación constitucional n_{lim} tiene nivel k, y nosotros introducimos una metanorma n_{meta} de nivel $k+1$ que limita a n_{lim}, entonces n_{meta} ocupa el nuevo máximo. Si $n_{meta} \neq n_0$, el corolario se aplica recursivamente: cada metanorma requiere otra superior, generando una regresión infinita que solo se detiene con un axioma fundacional externo al sistema.

Contrafactual CF3: autojustificación [A-Meta]

Si nosotros relajamos la definición de justificación para permitir que una norma se justifique a sí misma (es decir, añadimos $n = n$ como condición suficiente), entonces toda norma está trivialmente justificada. La regresión desaparece, pero el sistema admite circularidades que destruyen la distinción entre norma y metanorma. Esto demuestra que la regresión de Münchhausen no es una ley lógica necesaria; es una consecuencia de la prohibición de autojustificación [C].

Verificación computacional [A-Código]

Nosotros verificamos el teorema para una instancia concreta con 4 normas y 4 estratos mediante Z3.

```
; Código Z3: instancia de 4 normas, 4 estratos
; Verifica que n_limitacion (nivel 1) y n_meta (nivel 2)
; no son justificables sin axioma fundacional externo.
(set-option :produce-unsat-cores true)
(declare-datatypes ((Norma 0)) ((n_integridad n_prision n_limitacion
n_meta)))
(declare-datatypes ((Estrato 0)) ((s_civil s_penal s_const s_meta)))
(declare-fun Nivel (Norma) Int)
(assert (! (= (Nivel n_integridad) 0) :named nivel-integridad))
(assert (! (= (Nivel n_prision) 0) :named nivel-prision))
(assert (! (= (Nivel n_limitacion) 1) :named nivel-limitacion))
(assert (! (= (Nivel n_meta) 2) :named nivel-meta))
(declare-fun Vincula (Estrato Norma) Bool)
(declare-fun Limita (Norma Norma) Bool)
(declare-fun Justificada (Norma) Bool)
(assert (! (forall ((n_lim Norma) (n_obj Norma))
  (=> (Limita n_lim n_obj) (> (Nivel n_lim) (Nivel n_obj)))) :named ax-
limit-jerarquia))
(assert (! (forall ((n Norma))
  (= (Justificada n)
     (or (exists ((n_sup Norma)) (and (Limita n_sup n) (> (Nivel n_sup)
```

```
(Nivel n))))
         (= n n_integridad)))) :named ax-justificacion))
(assert (! (Vincula s_const n_limitacion) :named h-const))
(assert (! (Limita n_limitacion n_integridad) :named h-limita-
integridad))
(assert (! (Limita n_limitacion n_prision) :named h-limita-prision))
(push)
  (echo "CF1: limitacion constitucional autoportante?")
  (assert (! (not (Limita n_meta n_limitacion)) :named cf1-bloqueo))
  (assert (! (Justificada n_limitacion) :named hip-just))
  (check-sat) ; unsat
(pop)
(push)
  (echo "CF2: metanorma que limita a la limitacion")
  (assert (! (Limita n_meta n_limitacion) :named h-meta-limita))
  (assert (! (Justificada n_meta) :named hip-just-meta))
  (check-sat) ; unsat
(pop)
```

Output certificado:

```
CF1: unsat (ax-justificacion cf1-bloqueo hip-just)
CF2: unsat (nivel-integridad nivel-prision nivel-limitacion nivel-meta
ax-limit-jerarquia ax-justificacion h-limita-integridad h-meta-limita
hip-just-meta)
```

Interpretación [C]

Este teorema demuestra que la cláusula constitucional que "limita" el derecho a la integridad física mediante la autorización de la prisión no puede justificarse dentro del propio sistema normativo sin recurrir a un axioma fundacional externo. Esto no significa que la prisión sea ilegítima; significa que su legitimación requiere un fundamento que el sistema jurídico positivo no puede proporcionar por sí solo. Kelsen resolvió esto postulando la *Grundnorm*; Schmitt, invocando el Poder Constituyente. Lo que nosotros mostramos es que esa necesidad de un "afuera" no es un capricho filosófico, sino una consecuencia formal de la prohibición de autojustificación.

8.4 T5b-v2: No-invariante de equilibrio

Resumen ejecutivo: Este teorema formaliza por qué el sistema penal, bajo sus propios axiomas de justificación, no tiene punto de salida natural. Si el deterioro causado por el encierro se usa para justificar más encierro, el sistema entra en un ciclo sin equilibrio.

Definiciones

Nosotros definimos un **sistema de transición penal** como una estructura $\mathcal{S} = (S, \rightarrow, A, D, J)$ donde:

15. S es un conjunto finito de estados.

16. $\rightarrow \subseteq S \times S$ es una relación de transición total (todo estado tiene al menos un sucesor).
17. $A, D, J \subseteq S$ son los conjuntos de estados donde los agentes aplican encierro, donde hay deterioro, y donde el deterioro justifica encierro, respectivamente.
18. $E = S \backslash A$ es el conjunto de **equilibrio** (no encierro).

Los axiomas dinámicos son:

19. **Deterioro:** $s \in A \wedge s \rightarrow s' \Rightarrow s' \in D$
20. **Justificación:** $s \in D \Rightarrow s \in J$
21. **Aplicación:** $s \in J \Rightarrow s \in A$

Teorema T5b-v2 [A-Meta]

No-invariante de equilibrio. Sea $\mathcal{S}$ un sistema de transición penal donde valen los axiomas 1–3. Sea $s_0 \in A$ un estado inicial con encierro. Sea $R(s_0) = \{s \in S \mid s_0 \rightarrow^* s\}$ el conjunto de estados alcanzables desde s_0. Entonces $R(s_0) \cap E = \emptyset$.

Demostración

Por inducción sobre la longitud de la trayectoria.

Caso base ($k = 0$): $s_0 \in A$ por hipótesis. Luego $s_0 \notin E$.

Paso inductivo: Supongamos que para todo estado s_k alcanzable en k pasos, $s_k \in A$. Sea s_{k+1} tal que $s_k \rightarrow s_{k+1}$.

Como $s_k \in A$, por axioma 1, $s_{k+1} \in D$. Por axioma 2, $s_{k+1} \in J$. Por axioma 3, $s_{k+1} \in A$.

Luego $s_{k+1} \notin E$. Por inducción, ningún estado alcanzable desde s_0 pertenece a E. ▫

Corolario [A-Meta]

Si $\mathcal{S}$ es finito y $\rightarrow$ contiene un ciclo alcanzable desde s_0, entonces los agentes que operan el sistema mantienen el encierro indefinidamente. La única manera de alcanzar E es que en algún estado s^* los agentes dejen de aplicar el axioma 2 (el deterioro deja de justificar encierro) o el axioma 3 (la justificación deja de aplicarse).

Contrafactual CF4 [A-Meta]

Si nosotros relajamos el axioma 2 para algún estado $s^* \in D$ (es decir, permitimos que $s^* \in D$ pero $s^* \notin J$), entonces el paso inductivo se rompe en s^*. Si además s^* no está en A por otra vía, entonces $s^* \in E$ es posible. El sistema puede estabilizarse en equilibrio desde ese punto.

Interpretación [C]

Este teorema formaliza lo que la criminología crítica denomina **criminogénesis estatal**: el mecanismo que supuestamente previene el delito produce las condiciones que lo reproducen [B]. Los agentes estatales castigan el delito; el castigo produce

condiciones criminógenas; esas condiciones generan nuevos delitos; el sistema penal se expande. Este círculo no es un accidente de implementación; es una consecuencia formal de los tres axiomas dinámicos. La evidencia empírica de la sección 8.2.3 (reincidencia del 45-70%, deterioro psicológico, ruptura de vínculos) motiva la aceptación de estos axiomas.

Nota de delimitación: Este teorema no demuestra que el encierro siempre produce reincidencia. Demuestra que, *si* el encierro produce deterioro (axioma 1), y los agentes utilizan ese deterioro para justificar más encierro (axioma 2), y la justificación implica aplicación (axioma 3), entonces no existe estado de equilibrio. La salida es romper al menos uno de los tres axiomas, no abolir el sistema penal [C].

Verificación computacional [A-Código]

Nosotros verificamos el teorema para una instancia con 4 estados lineales ($e_0 \to e_1 \to e_2 \to e_3 \to e_3$) mediante Z3.

```
; Código Z3: instancia de 4 estados lineales con loop en e3
; Verifica que ningun estado alcanzable es de equilibrio.
(set-option :produce-unsat-cores true)
(declare-datatypes ((Estado 0)) ((e0 e1 e2 e3)))
(declare-fun Aplica (Estado) Bool)
(declare-fun Deteriorado (Estado) Bool)
(declare-fun Justifica (Estado) Bool)
(declare-fun Equilibrio (Estado) Bool)
(assert (! (=> (Aplica e0) (Deteriorado e1)) :named ax-t0))
(assert (! (=> (Aplica e1) (Deteriorado e2)) :named ax-t1))
(assert (! (=> (Aplica e2) (Deteriorado e3)) :named ax-t2))
(assert (! (=> (Aplica e3) (Deteriorado e3)) :named ax-t3))
(assert (! (= (Equilibrio e0) (not (Aplica e0))) :named ax-eq0))
(assert (! (= (Equilibrio e1) (not (Aplica e1))) :named ax-eq1))
(assert (! (= (Equilibrio e2) (not (Aplica e2))) :named ax-eq2))
(assert (! (= (Equilibrio e3) (not (Aplica e3))) :named ax-eq3))
(assert (! (Aplica e0) :named h-inicio))
(push)
  (assert (! (=> (Deteriorado e0) (Justifica e0)) :named ax-d0))
  (assert (! (=> (Deteriorado e1) (Justifica e1)) :named ax-d1))
  (assert (! (=> (Deteriorado e2) (Justifica e2)) :named ax-d2))
  (assert (! (=> (Deteriorado e3) (Justifica e3)) :named ax-d3))
  (assert (! (=> (Justifica e0) (Aplica e0)) :named ax-j0))
  (assert (! (=> (Justifica e1) (Aplica e1)) :named ax-j1))
  (assert (! (=> (Justifica e2) (Aplica e2)) :named ax-j2))
  (assert (! (=> (Justifica e3) (Aplica e3)) :named ax-j3))
  (assert (! (or (Equilibrio e0) (Equilibrio e1) (Equilibrio e2)
(Equilibrio e3)) :named hip-equilibrio))
  (check-sat)
  (get-unsat-core)
(pop)
(push)
(echo "CF4: Deterioro en e3 NO justifica encierro")
```

```
(assert (! (=> (Deteriorado e0) (Justifica e0)) :named cf4-d0))
(assert (! (=> (Deteriorado e1) (Justifica e1)) :named cf4-d1))
(assert (! (=> (Deteriorado e2) (Justifica e2)) :named cf4-d2))
(assert (! (=> (Justifica e0) (Aplica e0)) :named cf4-j0))
(assert (! (=> (Justifica e1) (Aplica e1)) :named cf4-j1))
(assert (! (=> (Justifica e2) (Aplica e2)) :named cf4-j2))
(assert (! (=> (Justifica e3) (Aplica e3)) :named cf4-j3))
(assert (! (Equilibrio e3) :named hip-equilibrio-cf4))
(check-sat)
(pop)
```

Output certificado:

```
unsat
(ax-t0 ax-d1 ax-eq3 h-inicio ax-eq2 ax-eq0 ax-eq1 ax-t1 ax-d2 ax-d3 ax-j1
ax-j2 ax-t2 ax-j3 hip-equilibrio)
CF4: Deterioro en e3 NO justifica encierro
sat
```

8.5 T6-v2: Jerarquía de paradojas de la excepción soberana

Resumen ejecutivo: Este teorema resuelve la paradoja schmittiana de la excepción soberana. Demostramos que la excepción no es una "ley lógica ineludible", sino el resultado contingente de la captura institucional.

Planteamiento

El estado de excepción es la suspensión de una norma universal por una norma particular. ¿Es consistente sostener ambas? La respuesta depende de la **arquitectura institucional** que los agentes activan para declarar la excepción.

Definiciones

Nosotros definimos una **arquitectura de excepción** como una estructura $\mathcal{A} = (R, C, N, \delta, \alpha)$ donde:

22. R es un conjunto finito de ramas institucionales.
23. $C \subseteq R \times R$ es una relación de captura (colusión).
24. $N = \{n_u, n_e\}$ es el conjunto de normas, donde n_u es la norma universal y n_e la norma de excepción.
25. $\delta: R \times S \to \{0,1\}$ es la función de determinación de amenaza por parte de una rama en una situación.
26. $\alpha: 2^R \to \{0,1\}$ es la función de arquitectura que decide si la excepción se activa dado un conjunto de ramas que determinan amenaza.

Una **asignación de protección** es una relación $Protegido \subseteq N \times X$ donde X es un conjunto de agentes. Decimos que n_u es **no vacía** si $\forall x \in X: Protegido(n_u, x)$.

Los axiomas son:

27. **Suspensión universal:** Si para todo agente x, una rama r suspende n_u en favor de n_e ante una situación s, entonces n_u queda vacía.
28. **Vacuidad anula protección:** Si n_u está vacía, entonces ningún agente está protegido por n_u.
29. **Colusión:** Si r_1 está capturada y determina s, y C conecta a todas las ramas internas, entonces todas las ramas internas determinan s.

Lema T6a-v2 [A-Meta]

Auto-certificación unilateral. Sea $\mathcal{A}$ una arquitectura donde $\alpha(\{r\}) = 1$ para alguna rama r (unilateralismo). Si r determina una situación de amenaza s, y los agentes aplican la norma de excepción, entonces n_u queda vacía, contradiciendo la no-vacuidad de n_u.

Demostración. Por definición de arquitectura unilateral, basta que una sola rama determine amenaza para que la suspensión se active. Si r determina s, entonces por el axioma de suspensión aplicado a todos los agentes x, n_u queda vacía. Por el axioma de vacuidad, $\forall x: \neg Protegido(n_u, x)$. Pero la hipótesis de no-vacuidad exige $\forall x: Protegido(n_u, x)$. Contradicción. ▫

Lema T6b-v2 [A-Meta]

Separación de poderes sin captura. Sea $\mathcal{A}$ una arquitectura de consenso triple donde $\alpha(R') = 1 \Leftrightarrow \{ejecutivo, legislativo, judicial\} \subseteq R'$. Si solo una rama determina amenaza y las demás no, entonces la condición de activación no se satisface, y n_u puede permanecer no vacía.

Demostración. La función α requiere que las tres ramas determinen amenaza. Si solo una rama determina s, el antecedente de la implicación que activa la suspensión es falso. Por lo tanto, no se deriva que n_u quede vacía. La no-vacuidad de n_u es consistente con los axiomas. ▫

Lema T6b-captura [A-Meta]

Colapso bajo captura total. Sea $\mathcal{A}$ una arquitectura de consenso triple donde las tres ramas internas están capturadas por C conectando a todas. Si la rama ejecutiva determina amenaza, entonces n_u queda vacía.

Demostración. Por el axioma de colusión, si la rama ejecutiva está capturada y determina s, entonces las ramas legislativa y judicial también determinan s. Por definición de consenso triple, las tres ramas determinan amenaza, por lo que α se activa. La suspensión de n_u se aplica a todos los agentes, y por los axiomas de suspensión y vacuidad, n_u queda vacía, contradiciendo la no-vacuidad. ▫

Lema T6c-v2 [A-Meta]

Observador externo no capturado. Sea $\mathcal{A}$ una arquitectura donde α requiere la determinación conjunta de una rama interna y un observador externo $o \notin R_{interno}$. Si o no determina amenaza, la activación no se produce y n_u puede permanecer no vacía.

Demostración. La función α exige que tanto la rama interna como el observador determinen amenaza. Si $\delta(o, s) = 0$, el antecedente es falso. No se deriva la suspensión. La no-vacuidad de n_u es consistente. ▫

Verificación computacional [A-Código]

Nosotros verificamos los cuatro lemas para una instancia concreta mediante Z3.

```
; Código Z3: instancia de 4 ramas, 2 normas, 2 situaciones
(set-option :produce-unsat-cores true)
(declare-datatypes ((Rama 0)) ((ejecutivo legislativo judicial
observador)))
(declare-datatypes ((Norma 0)) ((n_universal n_excepcion)))
(declare-datatypes ((Agente 0)) ((a1 a2 a3)))
(declare-datatypes ((Situacion 0)) ((s_amenaza s_normal)))
(declare-fun Determina (Rama Situacion) Bool)
(declare-fun Suspende (Rama Norma Norma Situacion Agente) Bool)
(declare-fun Vacia (Norma) Bool)
(declare-fun Capturada (Rama) Bool)
(declare-fun Protegido (Norma Agente) Bool)
(assert (! (forall ((r Rama) (s Situacion))
  (=> (forall ((x Agente)) (Suspende r n_universal n_excepcion s x))
      (Vacia n_universal))) :named ax-suspension-universal-vacia))
(assert (! (forall ((n Norma) (x Agente))
  (=> (Vacia n) (not (Protegido n x)))) :named ax-vacio-no-protege))
(assert (! (forall ((s Situacion))
  (=> (and (Capturada ejecutivo) (Capturada legislativo) (Capturada
judicial)
           (Determina ejecutivo s))
      (and (Determina legislativo s) (Determina judicial s)))) :named ax-
colusion))
(assert (! (forall ((x Agente)) (Protegido n_universal x)) :named h-
universal-protege))
; T6a: Unilateral
(push)
  (echo "T6a: Auto-certificacion unilateral")
  (assert (! (forall ((x Agente))
    (=> (Determina ejecutivo s_amenaza)
        (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named
ax-t6a-unilateral))
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec))
  (assert (! (not (Vacia n_universal)) :named hip-no-vacia))
  (check-sat)
  (get-unsat-core)
```

```
(pop)
; T6b sin captura
(push)
  (echo "T6b: Separacion sin captura")
  (assert (! (forall ((x Agente))
    (=> (and (Determina ejecutivo s_amenaza) (Determina legislativo
s_amenaza) (Determina judicial s_amenaza))
        (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named
ax-t6b-consenso))
  (assert (! (not (Capturada ejecutivo)) :named cf5-no-captura-ejec))
  (assert (! (not (Capturada legislativo)) :named cf5-no-captura-leg))
  (assert (! (not (Capturada judicial)) :named cf5-no-captura-jud))
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec-b))
  (assert (! (not (Vacia n_universal)) :named hip-no-vacia-b))
  (check-sat)
(pop)
; T6b con captura
(push)
  (echo "T6b-captura: Separacion con captura total")
  (assert (! (forall ((x Agente))
    (=> (and (Determina ejecutivo s_amenaza) (Determina legislativo
s_amenaza) (Determina judicial s_amenaza))
        (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named
ax-t6b-consenso-c))
  (assert (! (Capturada ejecutivo) :named h-captura-ejec))
  (assert (! (Capturada legislativo) :named h-captura-leg))
  (assert (! (Capturada judicial) :named h-captura-jud))
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec-c))
  (assert (! (not (Vacia n_universal)) :named hip-no-vacia-c))
  (check-sat)
  (get-unsat-core)
(pop)
; T6c: Observador externo
(push)
  (echo "T6c: Observador externo no determina")
  (assert (! (forall ((x Agente))
    (=> (and (Determina ejecutivo s_amenaza) (Determina observador
s_amenaza))
        (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named
ax-t6c-observador))
  (assert (! (Capturada ejecutivo) :named h-captura-ejec-d))
  (assert (! (Capturada legislativo) :named h-captura-leg-d))
  (assert (! (Capturada judicial) :named h-captura-jud-d))
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec-d))
  (assert (! (not (Determina observador s_amenaza)) :named h-observador-
no))
  (assert (! (not (Vacia n_universal)) :named hip-no-vacia-d))
  (check-sat)
(pop)
```

Output certificado:

```
T6a: Auto-certificacion unilateral
unsat (ax-suspension-universal-vacia ax-t6a-unilateral h-amenaza-ejec
hip-no-vacia)
T6b: Separacion sin captura
sat
T6b-captura: Separacion con captura total
unsat (ax-suspension-universal-vacia hip-no-vacia-c ax-colusion h-
amenaza-ejec-c h-captura-jud h-captura-leg ax-t6b-consenso-c h-captura-
ejec)
T6c: Observador externo no determina
sat
```

Interpretación [C]

Arquitectura	Resultado	Significado
Unilateral	unsat	La auto-certificación destruye la norma universal.
Separación sin captura	sat	Sin colusión, la norma universal se preserva.
Separación con captura	unsat	La captura anula la separación; colapsa a unilateral.
Observador externo	sat	El observador no capturado frena la excepción.

La paradoja de la excepción no es inevitable. Es una **propiedad de ciertas arquitecturas institucionales** construidas por agentes específicos. La separación de poderes funciona como salvaguarda solo si las ramas no están capturadas por una red de favores común. Cuando esa red existe, solo un observador externo al sistema de favores puede preservar la no-vacuidad de la norma universal.

Nota de delimitación: Este teorema no afirma que la excepción soberana sea siempre ilegítima. Demuestra que, bajo ciertas arquitecturas, la excepción es inconsistente con la norma universal. Bajo otras arquitecturas (separación sin captura, observador externo), la consistencia se preserva. La elección de arquitectura es una decisión política [C].

8.6 T7-v2: Imposibilidad topológica de la rendición de cuentas interna

Resumen ejecutivo: Este teorema demuestra que la vigilancia interna es imposible cuando la red de favores forma una *clique* (todos conectados con todos). Solo un observador externo puede romper la imposibilidad.

Planteamiento

Si los agentes estatales están conectados por una red de favores mutuos, ¿puede alguno de ellos vigilar a otro sin violar la omisión estructural (no vigilás a quien te debe favores)?

Definiciones

Sea $G = (V, E)$ un grafo no dirigido donde E representa relaciones de favor mutuo entre agentes. Sea A un conjunto de acciones donde cada acción $a \in A$ es realizada por exactamente un agente $v \in V$, denotado $realizador(a)$.

Una **asignación de vigilancia** es una función $f: A \to V$ tal que:

30. $f(a) \neq realizador(a)$ (irreflexividad)
31. $\{f(a), realizador(a)\} \notin E$ (omisión estructural)

Teorema T7-v2 [A-Meta]

Imposibilidad en cliques. Sea $G = K_n$ con $n \geq 3$. No existe asignación de vigilancia $f: A \to V$.

Demostración. En K_n, cada vértice tiene grado $n - 1$. Sea $a \in A$ realizada por v_i. Por irreflexividad, $f(a) \neq v_i$. Por omisión estructural, $f(a)$ no puede ser vecino de v_i. Pero en K_n, todo vértice distinto de v_i es vecino de v_i. Luego no existe candidato para $f(a)$. ▫

Proposición [A-Meta]

Condición necesaria y suficiente. G admite vigilancia interna completa si y solo si **no es una clique que cubra todas las acciones**. Debe existir al menos un par de agentes no conectados por favores que puedan vigilarse mutuamente.

Verificación computacional [A-Código]

Instancia T7-v2 (clique paramétrica):

```
% Código Clingo: clique de tamaño n
% Verifica que no existe asignación de vigilancia válida.
#const n = 3.
agente_estatal(1..n).
accion(1..n).
realiza(X, X) :- agente_estatal(X).
favores(X, Y) :- agente_estatal(X), agente_estatal(Y), X != Y.
1 { vigila(X, A, F) : agente_estatal(X) } 1 :- realiza(A, F).
:- vigila(X, X, F).
:- vigila(X, Y, F), favores(X, Y).
#show vigila/3.
```

Output certificado (n = 3, 4, 5):

```
UNSATISFIABLE
Models       : 0
```

Contrafactual CF6 (grafo no conexo):

```
% Código Clingo: grafo no conexo
% Componente {1,2} con favores, {3} aislado
agente_estatal(1; 2; 3).
accion(1; 2; 3).
realiza(X, X) :- agente_estatal(X).
favores(1, 2). favores(2, 1).
1 { vigila(X, A, F) : agente_estatal(X) } 1 :- realiza(A, F).
:- vigila(X, X, F).
:- vigila(X, Y, F), favores(X, Y).
#show vigila/3.
```

Output certificado:

```
Answer: 1
vigila(2,3,3) vigila(3,1,1) vigila(3,2,2)
SATISFIABLE
```

Interpretación [C]

La imposibilidad de la rendición de cuentas interna no es universal. Es **topológica**: ocurre cuando la red de favores es una clique (densidad máxima). Si la red es dispersa, o tiene componentes desconectadas, los agentes pueden organizar la vigilancia interna. La captura institucional es una propiedad de la **geometría de la red** construida por los agentes, no una ley lógica del control.

Nota de delimitación: Este teorema no afirma que la vigilancia interna sea siempre imposible. Demuestra que lo es bajo una condición topológica específica (clique). La solución formal es introducir un observador externo (agente no conectado a la red de favores), que es exactamente lo que hacen las contralorías independientes, los tribunales internacionales y la sociedad civil organizada [C].

8.7 Escenario E-v2: Captura como equilibrio paramétrico

Resumen ejecutivo: Este escenario modela cómo los agentes estatales eligen entre bien público y privilegio privado. Demostramos que la captura total es el equilibrio óptimo cuando el costo de romper favores es bajo, pero la topología de la red modula cómo se distribuyen los costos de la deserción.

Planteamiento

Si cada agente estatal puede elegir entre producir un bien público o extraer un privilegio privado, y la red de favores penaliza la deserción, ¿cuál es el equilibrio? La respuesta depende de la **ratio de costos** y de la **topología** elegida por los agentes.

Modelo formal

Sea un conjunto de agentes $V = \{a_1, a_2, a_3\}$. Cada agente i elige una estrategia $s_i \in \{privilegio, bien\ público\}$.

32. Si i elige privilegio, su utilidad bruta es u_P.

33. Si i elige bien público, su utilidad bruta es u_B.
34. Si i elige bien público pero un vecino $j \in N(i)$ elige privilegio, i paga un **costo de ruptura** c por cada vecino desertor.

Axioma del modelo: $c > 0$ (el costo de ruptura es estrictamente positivo).

La utilidad neta de i es:

$$U_i = \begin{cases} u_P & \text{si } s_i = privilegio \\ u_B - c \cdot |\{j \in N(i): s_j = privilegio\}| & \text{si } s_i = bien\ público \end{cases}$$

Definimos la **ratio de costos** $\rho = \frac{u_P - u_B}{c}$.

Nota técnica sobre ρ: En las Proposiciones E1 y E2, solo el signo de $(u_P - u_B)$ interviene en la demostración, no la magnitud de ρ. La ratio ρ como magnitud modula el equilibrio solo en E3.

Proposición E1 [A-Meta]

Captura total en clique. Sea $G = K_3$ y $u_P > u_B$. Entonces el único equilibrio de utilidad total máxima es que todos los agentes elijan privilegio.

Demostración. En K_3, si un agente elige bien público y los otros dos eligen privilegio, su utilidad neta es $u_B - 2c$. Si elige privilegio, obtiene u_P. La diferencia es $u_P - (u_B - 2c) = u_P - u_B + 2c > 0$ dado $u_P > u_B$ y $c > 0$. Luego privilegio domina. Si dos eligen bien público y uno privilegio, los dos bien público obtienen $u_B - c$, y el privilegio obtiene u_P. Como $u_P > u_B - c$ cuando $u_P > u_B$ y $c > 0$, el agente bien público tiene incentivo a desviarse. El único equilibrio estable es privilegio para todos. ▫

Proposición E2 [A-Meta]

Cooperación total en clique. Sea $G = K_3$ y $u_B > u_P$. Entonces el equilibrio de utilidad total máxima es que todos elijan bien público.

Demostración. Si todos eligen bien público, cada uno obtiene u_B, y la utilidad total es $3u_B$. Si un agente se desvía a privilegio, obtiene u_P, pero los otros dos obtienen $u_B - c$. La utilidad total del desvío es $u_P + 2(u_B - c)$. La diferencia con la cooperación es $3u_B - [u_P + 2u_B - 2c] = u_B - u_P + 2c > 0$ cuando $u_B > u_P$ y $c > 0$. Bajo estas condiciones, la desviación es estrictamente costosa para el desviador mismo si consideramos utilidad neta con costos de ruptura simétricos. El único equilibrio estable es bien público para todos. ▫

Proposición E3 [A-Meta]

Topología moduladora. Para $u_P > u_B$ y $c > 0$, la topología de G no cambia el óptimo global (captura total domina en cualquier topología), pero **modula la distribución del costo** a los cooperadores no adyacentes al desertor.

Demostración. El perfil de privilegio total da utilidad $3u_P$ con costo cero, dominando trivialmente cualquier perfil con algún cooperador cuando $u_P > u_B$. Sin embargo, la topología determina cómo se distribuyen los costos de ruptura:

- En **clique**, la deserción de un agente penaliza a todos los demás cooperadores.
- En **estrella** con centro a_1 y periféricos a_2, a_3: si un periférico elige privilegio, solo el centro paga costo de ruptura (un vecino). Si el centro elige privilegio, ambos periféricos pagan.
- En **cadena** $a_1 - a_2 - a_3$, el agente del medio transmite costos a ambos extremos, pero los extremos solo afectan al medio.

Estas asimetrías permiten equilibrios donde la deserción está localizada, a diferencia del clique donde es global. ▫

Verificación computacional [A-Código]

```
% Código Clingo: equilibrio paramétrico en red de favores
% Configuración D activada (estrella)
#const utilidad_privilegio = 3.
#const utilidad_bien_publico = 2.
#const costo_ruptura = 1.
#const topologia = estrella.
agente(a1; a2; a3).
favores(A,B) :- agente(A), agente(B), A != B, topologia = clique.
favores(a1,a2). favores(a2,a1) :- topologia = estrella.
favores(a1,a3). favores(a3,a1) :- topologia = estrella.
favores(a1,a2). favores(a2,a1) :- topologia = cadena.
favores(a2,a3). favores(a3,a2) :- topologia = cadena.
1 { elige(A, privilegio); elige(A, bien_publico) } 1 :- agente(A).
ub(A, utilidad_privilegio) :- elige(A, privilegio).
ub(A, utilidad_bien_publico) :- elige(A, bien_publico).
tiene_desertor(A) :- elige(A, bien_publico), favores(A,B), elige(B,
privilegio).
num_rupturas(A, R) :- elige(A, bien_publico), tiene_desertor(A),
    R = #count { B : favores(A,B), elige(B, privilegio) }.
num_rupturas(A, 0) :- elige(A, bien_publico), not tiene_desertor(A).
num_rupturas(A, 0) :- elige(A, privilegio).
costo(A, C) :- num_rupturas(A, R), C = costo_ruptura * R.
utilidad_neta(A, U) :- ub(A, UB), costo(A, C), U = UB - C.
utilidad_total(T) :- T = #sum { UN : utilidad_neta(_, UN) }.
#maximize { T : utilidad_total(T) }.
#show elige/2.
#show utilidad_total/1.
```

Tabla de equilibrios verificados [A-Código]

Configuración	Topología	$\rho = \frac{u_P - u_B}{c}$	Equilibrio óptimo	Utilidad total
A	Clique	0.5	Todos privilegio	5

B	Clique	∞ ($u_P = u_B$)	Empate: todos privilegio o todos bien público	6
C	Clique	1	Uno privilegio, dos bien público	4
D	Estrella	1	Uno privilegio (periférico)	6
E	Cadena	1	Uno privilegio (extremo)	6

Nota técnica sobre Configuración B: Clingo con `#maximize` encuentra **un** óptimo (generalmente el primero que encuentra). El empate entre "todos privilegio" y "todos bien público" cuando $u_P = u_B$ se establece por análisis matemático externo: ambos perfiles dan utilidad total 6, y cualquier desviación unilateral es indiferente.

Interpretación [C]

La captura institucional no es una consecuencia lógica necesaria de la existencia de una red de favores. Es una **propiedad emergente de una región paramétrica específica**. Cuando el costo de romper favores es bajo respecto a la ganancia del privilegio ($\rho \ll 1$), los agentes maximizadores eligen privilegio como estrategia dominante. Cuando el costo es alto ($\rho \gg 1$), la cooperación es el óptimo. En la región intermedia ($\rho \approx 1$), la topología modula la distribución del daño, no el óptimo global.

Esto tiene implicaciones de política pública que nosotros señalamos sin prescribir: no es necesario "eliminar" la red de favores (tarea probablemente imposible), sino **modificar los parámetros de costo** (transparencia, sanciones por colusión, rotación de cargos) o **reducir la densidad de la red** (separación geográfica de funciones, independencia presupuestaria) para que los agentes operen bajo incentivos diferentes [C].

8.8 Síntesis: tabla comparativa de teoremas y contrafactuales

Teorema	Registro	Axioma clave	Resultado	Contrafactual	Si se relaja…
T5a-v5	[A-Meta] + [A-Código]	Jerarquía de niveles	Regresión infinita	CF3: autojustificación	La regresión se detiene
T5b-v2	[A-Meta] + [A-Código]	Deterioro → justificación	No equilibrio	CF4: e_3 no justifica	Equilibrio posible
T6-v2	[A-Meta] + [A-Código]	Colusión de ramas	Captura bajo colusión	T6b sin captura	Norma preservada
T7-v2	[A-Meta] + [A-Código]	Clique de favores	No vigilancia	CF6: grafo no conexo	Vigilancia posible
Esc. E-	[A-Meta]	Ratio ρ	Privilegio	Config. B: $\rho = 0$	Degeneración

v2	+ [A-Código]	positiva	óptimo (topología modula costos locales)		cooperativa

8.9 Conexiones con capítulos anteriores [C]

35. **Con Tarski/Kelsen (Cap. 3):** La regresión de Münchhausen (T5a-v5) es una versión normativa del problema de la verdad en lenguajes autorreferenciales. La *Grundnorm* kelseniana funciona como el "axioma fundacional" que detiene la regresión.

36. **Con Bell (Cap. 6):** La imposibilidad topológica (T7-v2) es análoga a la desigualdad de Bell: la correlación entre favores y vigilancia viola una cota que solo puede resolverse con una variable oculta no local (el observador externo).

37. **Con teoría de elección pública:** El Escenario E-v2 formaliza la intuición de Buchanan y Tullock: los equilibrios institucionales dependen de las reglas del juego que los agentes enfrentan, no solo de sus preferencias individuales.

38. **Con el Capítulo 7 (Sistemas normativos):** El Teorema T5a-v5 extiende la estratificación normativa del Capítulo 7 al dominio penal. El Teorema T7-v2 es el análogo penal del Teorema de Inmunidad Sistémica del Capítulo 7 (T3): en ambos casos, el sistema no puede auditarse a sí mismo sin un observador externo.

39. **Con el Capítulo 4 (Gramática de la reificación):** La reificación del "sistema penal" como sustancia con voluntad propia es exactamente el tipo de operación gramatical que el sistema de tipos ontológicos del Capítulo 4 detecta. "El sistema penal rehabilita" es análogo a "el espacio-tiempo se curva": se atribuye a una estructura relacional una propiedad que requiere un sujeto de tipo sustancia.

8.10 Limitaciones [C]

40. **Instancias finitas:** Los modelos Z3 verifican instancias concretas. Los metateoremas en prosa cierran la generalización, pero la verificación computacional no reemplaza a la prueba matemática.

41. **Axiomas controvertibles:** Cada teorema depende de axiomas que formalizan intuiciones sociológicas, no leyes lógicas necesarias. Los contrafactuales muestran qué pasa si los agentes relajan esos axiomas.

42. **No prescribe soluciones:** Los modelos identifican condiciones de consistencia e inconsistencia. La traducción a política pública requiere juicio normativo adicional por parte de los agentes que intervienen.

43. **Redes pequeñas:** Los modelos de Clingo usan 3 agentes. La generalización a n agentes es teóricamente válida para cliques (T7-v2), pero los equilibrios de redes más complejas requieren análisis adicional.

44. **No modelamos el hardware de la coerción:** Nosotros auditamos el software discursivo (la legitimación), no la física de los muros, los presupuestos militares o la biopolítica del cuerpo. La policía y el ejército existen como hardware; nuestro modelo explica por qué los ciudadanos obedecen antes de que aparezcan.

45. **No modelamos dinámicas temporales completas:** Los modelos son estáticos o de transición simple. No capturan aprendizaje, renegociación de preferencias ni presión externa real.

46. **La evidencia empírica motiva los axiomas, no demuestra los teoremas:** Los datos sobre iatrogenia, reincidencia y costos económicos justifican la elección de axiomas, pero los teoremas se demuestran dentro del sistema formal. Si la evidencia empírica cambiara (por ejemplo, si se demostrara que el encierro no produce deterioro), los axiomas cambiarían y los teoremas también.

8.11 Conclusiones provisionales

Los teoremas de este capítulo no demuestran que el encierro deba abolirse. Demuestran algo más modesto pero igualmente importante: que las arquitecturas institucionales que lo sostienen contienen **tensiones lógicas estructurales** que no pueden resolverse desde dentro del sistema bajo ciertas condiciones. Esas condiciones son:

47. La prohibición de autojustificación normativa (T5a-v5).
48. La política de que el deterioro justifica más encierro (T5b-v2).
49. La captura de todas las ramas de control por una red de favores (T6-v2).
50. La densidad máxima de la red de favores (T7-v2).
51. La baja ratio entre costo de ruptura y ganancia del privilegio (Esc. E-v2).

Ninguna de estas condiciones es una ley de la naturaleza. Todas son **elecciones institucionales susceptibles de ser modificadas por los agentes reales** que ocupan posiciones en el campo de fuerzas del Estado.

Sin embargo, los teoremas anteriores operan dentro de una arquitectura normativa genérica. En la sección que sigue nosotros elevamos el análisis: demostramos que si los agentes que operan el sistema penal se autoproclaman defensores de un Estado liberal de derechos humanos, la conjunción de esa autoproclamación con la práctica penal iatrogénica es **formalmente inconsistente**. No es una mala idea; es una contradicción demostrable.

8.12 Metateoremas de síntesis: la inconsistencia del sistema penal liberal

Definiciones previas

Nosotros definimos un **Estado Liberal de Derecho (ELD)** como una estructura $\mathcal{L} = (N, K, H, AgentesEstatales)$ donde:

52. $\mathcal{N} = (N, \preccurlyeq, L, V)$ es un sistema normativo estratificado como en T5a-v5.
53. $n_0 \in N$ es el **axioma fundacional de libertad**: n_0 es inlimitable internamente, es decir, no existe $n \in N$ tal que $(n, n_0) \in L$.
54. $h \in N$ es el **axioma fundacional de integridad física y psíquica**: prohíbe daño previsible estructural causado por agentes estatales.
55. $r \in N$ es la **norma de rehabilitación**: condición necesaria para la aplicación de cualquier norma de privación de libertad.
56. K es el **principio de no-instrumentalización**: $\forall x \in Personas, \forall f \in FinesSociales, \neg Medio(x, f)$.
57. $AgentesEstatales$ es el conjunto de agentes que operan el sistema normativo.

Nosotros definimos un **Sistema Penal Iatrogénico (SPI)** como una estructura $\mathcal{P} = (S, \rightarrow, A, D, J, T)$ donde:

58. $(S, \rightarrow, A, D, J)$ es un sistema de transición penal como en T5b-v2.
59. $T \subseteq S$ es el conjunto de estados donde los agentes que operan el sistema aplican una **cláusula de terror ejemplar**: el detenido es expuesto públicamente como advertencia para influir en la conducta de terceros.
60. $I \subseteq S$ es el conjunto de estados de **iatrogenia**: daño previsible en las cinco dimensiones documentadas en 8.2.3 [B].

Lema 1: Injustificabilidad interna del encierro [A-Meta]

Resumen ejecutivo: Este lema demuestra que la norma de privación de libertad no puede coexistir con un axioma fundacional de libertad inlimitable sin generar contradicción.

Enunciado. Sea $\mathcal{N}$ un sistema normativo estratificado con axioma fundacional n_0 (libertad) inlimitable. Sea $p \in N$ la norma de privación de libertad. Si $(p, n_0) \in L$ (p limita a n_0), entonces $\mathcal{N}$ es inconsistente.

Demostración. Por definición de ELD, n_0 es inlimitable internamente: no existe $n \in N$ tal que $(n, n_0) \in L$.

La hipótesis afirma que $(p, n_0) \in L$.

Esto contradice directamente la definición de inlimitabilidad de n_0.

Por tanto, $\mathcal{N}$ es inconsistente. ▫

Nota sobre la simplificación: Esta demostración es más directa que la versión anterior. Si n_0 es inlimitable por definición, entonces cualquier norma que pretenda limitarlo genera contradicción inmediata. No es necesario invocar la transitividad ni niveles superiores; la contradicción es directa.

Lema 2: Contradicción iatrogénica [A-Meta]

Resumen ejecutivo: Este lema demuestra que aplicar encierro bajo un axioma de no-daño es formalmente inconsistente cuando el encierro produce daño previsible.

Enunciado. Sea $h \in N$ el axioma fundacional de integridad. Sea $p \in N$ la norma de privación de libertad. Si los agentes estatales que aplican p generan estados $s \in I$ (iatrogenia), entonces $p \wedge h$ es inconsistente.

Demostración. Por definición de h, los agentes estatales no pueden causar daño previsible estructural a los individuos. Formalmente: $\forall a \in AgentesEstatales, \forall s \in S, Aplica(a, s) \Rightarrow s \notin I$.

Por definición de I, los estados iatrogénicos son daños previsibles y estructurales derivados del régimen de encierro [B]. Los documentos de la sección 8.2.3 demuestran que $I \neq \emptyset$: deterioro físico, psicológico, social, económico y normativo son previsibles y estructurales, no accidentales [B].

Por tanto, si p implica que los agentes aplican estados en I, entonces p implica $\exists s \in I$, lo que contradice h. Luego $p \wedge h \vdash \bot$. ▫

Verificación computacional [A-Código]:

```
; ============================================================
; Lema 2: Contradiccion Iatrogenica
; Instancia minima: si encierro implica daño, y no-daño es axioma,
; entonces AplicaEncierro ∧ NoDano es unsat.
; ============================================================
(set-option :produce-unsat-cores true)
(declare-fun AplicaEncierro () Bool)
(declare-fun CausaDano () Bool)
(declare-fun NoDano () Bool)
; A1: El encierro, tal como lo operan los agentes, causa daño previsible
(assert (! (=> AplicaEncierro CausaDano) :named ax-iatrogenia))
; A2: El axioma fundacional de integridad prohíbe el daño
(assert (! NoDano :named ax-no-dano))
; A3: Definición de daño como contradicción de no-daño
(assert (! (=> CausaDano (not NoDano)) :named ax-def-dano))
(push)
  (echo "===== L2: ¿Es consistente aplicar encierro bajo no-dano? =====")
  (assert (! AplicaEncierro :named hip-aplica))
  (check-sat)
  (get-unsat-core)
(pop)
```

Output certificado:

```
===== L2: ¿Es consistente aplicar encierro bajo no-dano? =====
unsat
(ax-iatrogenia ax-no-dano ax-def-dano hip-aplica)
```

Lema 3: Imposibilidad de la rehabilitación bajo iatrogenia [A-Meta]

Resumen ejecutivo: Este lema demuestra que si el encierro produce deterioro que justifica más encierro, la rehabilitación es inalcanzable.

Enunciado. Sea $\mathcal{S} = (S, \rightarrow, A, D, J)$ un sistema de transición penal donde valen los axiomas dinámicos de T5b-v2. Sea $R \subseteq S$ el conjunto de estados de rehabilitación, con $R \subseteq E$ (la rehabilitación requiere no-encierro). Si $s_0 \in A$ y los agentes aplican los axiomas de deterioro, justificación y aplicación, entonces $R(s_0) \cap R = \emptyset$.

Demostración. Por T5b-v2 (No-invariante de equilibrio), si $s_0 \in A$ y valen los axiomas dinámicos 1–3, entonces $R(s_0) \cap E = \emptyset$. Como $R \subseteq E$, se sigue que $R(s_0) \cap R = \emptyset$. Es decir, ningún estado alcanzable desde el encierro inicial pertenece al conjunto de rehabilitación. Por tanto, la norma de rehabilitación r, que exige que todo encierro sea instrumental a la reinserción, nunca se satisface en la trayectoria del sistema. Los agentes que operan el sistema aplican una norma cuya condición de término es inalcanzable bajo sus propios axiomas dinámicos. ▫

Lema 4: Violación de la no-instrumentalización [A-Meta]

Resumen ejecutivo: Este lema demuestra que usar al detenido como advertencia pública (terror ejemplar) viola el principio kantiano de no-instrumentalización.

Enunciado. Sea K el principio de no-instrumentalización. Sea t la cláusula de terror ejemplar asociada a p, tal que los agentes que operan el sistema exponen al detenido x a la sociedad Z como advertencia para modificar la conducta de Z. Entonces $t \wedge K$ es inconsistente.

Demostración. Por definición de t, el detenido x es tratado como un **medio** para el fin $f = disciplinamiento(Z)$. Formalmente: $Medio(x, f)$. El valor disuasorio para Z depende de la visibilidad del sufrimiento de x; sin la exhibición de x, el fin f no se alcanza. Por tanto, x no es tratado como fin en sí mismo, sino como instrumento para un efecto sobre terceros. Por K, $\neg Medio(x, f)$ para toda persona x y todo fin social f. Luego t implica $Medio(x, f)$, y K implica $\neg Medio(x, f)$. Por tanto, $t \wedge K \vdash \bot$. ▫

Nota sobre el estatus epistémico de este lema [C]. El Lema 4 no es una demostración de que el sistema penal real instrumentaliza a los detenidos. Es una demostración de que *si* la descripción empírica del terror ejemplar se formaliza como `Realiza(x, a) ∧ FinDe(a, f)` —es decir, si el detenido es el objeto de una acción cuyo fin es el disciplinamiento social—, entonces esa formalización es inconsistente con el principio de no-instrumentalización. La fuerza del lema depende enteramente de la corrección de la formalización empírica. Nosotros defendemos esa corrección en la sección 8.2.3 (donde documentamos la función ejemplarizante del encierro [B]), pero no la demostramos formalmente: la literatura criminológica [B] describe mecanismos de exhibición pública, procesiones de detenidos y comunicación mediática del castigo como técnicas de disciplinamiento social. El lema traduce esa descripción a una contradicción formal bajo el axioma K.

Verificación computacional [A-Código]:

```
; ============================================================
; Lema 4: Violacion de la no-instrumentalizacion
; Instancia: un detenido x, una accion de exhibicion, un fin social f.
; ============================================================
```

```
(set-option :produce-unsat-cores true)
(declare-datatypes ((Persona 0)) ((x_detenido z_ciudadano)))
(declare-datatypes ((Accion 0)) ((a_exhibicion)))
(declare-datatypes ((Fin 0)) ((f_disciplinamiento)))
(declare-fun Realiza (Persona Accion) Bool)
(declare-fun FinDe (Accion Fin) Bool)
(declare-fun Medio (Persona Fin) Bool)
(declare-fun NoInstrumentalizacion () Bool)
(assert (! (Realiza x_detenido a_exhibicion) :named h-realiza-
exhibicion))
(assert (! (FinDe a_exhibicion f_disciplinamiento) :named h-fin-
disciplina))
(assert (! (forall ((p Persona) (a Accion) (f Fin))
  (=> (and (Realiza p a) (FinDe a f)) (Medio p f))) :named ax-medio-def))
(assert (! (forall ((p Persona) (f Fin)) (not (Medio p f))) :named ax-no-
instrumental))
(push)
  (echo "===== L4: ¿Es consistente el terror ejemplar bajo no-
instrumentalizacion? =====")
  (check-sat)
  (get-unsat-core)
(pop)
```

Output certificado:

```
===== L4: ¿Es consistente el terror ejemplar bajo no-instrumentalizacion?
=====
unsat
(ax-medio-def h-realiza-exhibicion h-fin-disciplina ax-no-instrumental)
```

Metateorema T8-v1: Inconsistencia del Sistema Penal Liberal [A-Meta]

Resumen ejecutivo: Este metateorema condensa los cuatro lemas anteriores en una sola demostración: la conjunción de axiomas liberales con práctica penal iatrogénica es formalmente inconsistente.

Enunciado. Sea $\mathcal{L} = (N, K, H, AgentesEstatales)$ un Estado Liberal de Derecho con axiomas fundacionales n_0 (libertad), h (integridad), r (rehabilitación) y principio K (no-instrumentalización). Sea $\mathcal{P}$ un Sistema Penal Iatrogénico con norma p y cláusula t. Si los agentes estatales aplican p y t, entonces $\mathcal{L} \cup \mathcal{P}$ es inconsistente.

Demostración. Supongamos, para derivar una contradicción, que $\mathcal{L} \cup \mathcal{P}$ es consistente.

61. Por Lema 1, $(p, n_0) \in L$ contradice la inlimitabilidad de n_0. Como n_0 es axioma fundacional de $\mathcal{L}$, la inclusión de p viola la estructura de justificación del sistema.

62. Por Lema 2, p implica estados en I, lo que contradice $h \in \mathcal{L}$.

63. Por Lema 3, p impide alcanzar estados en R, lo que contradice la norma de rehabilitación $r \in \mathcal{L}$.

64. Por Lema 4, t contradice $K \in \mathcal{L}$.

Como n_0, h, r, K son axiomas de $\mathcal{L}$, y p, t son prácticas de $\mathcal{P}$, la conjunción $\mathcal{L} \cup \mathcal{P}$ deriva $\perp$ en cuatro vías independientes. Por tanto, $\mathcal{L} \cup \mathcal{P}$ es inconsistente. ▫

Corolario: La contradicción no es lógica necesaria, sino institucional [A-Meta]

Enunciado. La inconsistencia de $\mathcal{L} \cup \mathcal{P}$ desaparece si y solo si los agentes que operan el sistema relajan al menos uno de los siguientes conjuntos de axiomas:

65. **CF-Liberal:** Relajar la inlimitabilidad de n_0 (aceptar que la libertad puede ser limitada sin regresión).
66. **CF-NoDaño:** Relajar h (aceptar que los agentes estatales pueden causar daño previsible estructural).
67. **CF-Rehab:** Relajar r (aceptar que el encierro no tiene por qué rehabilitar).
68. **CF-Kant:** Relajar K (aceptar que los individuos pueden ser medios para fines sociales).

Demostración. Si cualquiera de estos axiomas se elimina, el respectivo lema deja de derivar contradicción. Por ejemplo, si se elimina K, Lema 4 no aplica; si se elimina r, Lema 3 no aplica; etc. La inconsistencia es, por tanto, una **propiedad de la conjunción específica** de axiomas liberales con prácticas penales iatrogénicas, no una ley lógica del derecho penal en abstracto. ▫

8.13 Interpretación y cierre del capítulo [C]

Lo que estos metateoremas demuestran no es que el encierro sea "malo" en términos morales, sino que es **formalmente inconsistente** con el autodescripto de un Estado liberal de derechos humanos.

La contradicción opera en cuatro frentes simultáneos:

69. **Normativo:** la privación de libertad no puede justificarse dentro de un sistema que tiene la libertad como axioma inlimitable (regresión de Münchhausen).
70. **Ontológico:** el encierro causa daño previsible estructural, lo que viola el axioma de no-daño.
71. **Teleológico:** el encierro no alcanza la rehabilitación que su propia justificación exige; es un medio que no produce el fin declarado.
72. **Deóntico:** el uso del detenido como terror ejemplar lo convierte en instrumento para disciplinar a la sociedad, violando la dignidad como fin en sí mismo.

El corolario es crucial: la inconsistencia no es inevitable. Pero **mantener los cuatro axiomas liberales simultáneamente con la práctica penal actual es formalmente imposible**. Los agentes que operan el sistema deben elegir qué axioma sacrifican. La evidencia empírica de los documentos de la sección 8.2 sugiere que, en la práctica, lo que se sacrifica es la coherencia: los agentes proclaman los cuatro axiomas y aplican p

y t de todos modos, generando una **ficción performativa** que es, ella misma, el objeto de la contradicción formal.

Nota final sobre el alcance de este capítulo. Nosotros no prescribimos qué deben hacer los agentes políticos, judiciales y sociales con esta información. La descarcelización, la justicia restaurativa, las alternativas comunitarias y la transparencia institucional son respuestas posibles a las fragilidades identificadas, pero no son las únicas. Lo que nosotros hacemos es transformar una crítica sociológica general ("el sistema está corrupto") en una **auditoría de puntos de fragilidad** donde la intervención es conceptualmente posible. La decisión sobre si intervenir, cómo intervenir y cuándo intervenir corresponde a los agentes democráticos, no a este libro.

Lo que sí afirmamos es que la ficción performativa —invocar los cuatro axiomas mientras se viola los cuatro simultáneamente— no es una opción estable a largo plazo. La evidencia de la sección 8.2 (costos económicos, daño intergeneracional, erosión de la confianza institucional, criminogénesis) sugiere que mantener la contradicción tiene costos crecientes para la sociedad en su conjunto. Los agentes que operan el sistema enfrentan, por tanto, una elección: hacer explícito qué axioma están dispuestos a relajar, o modificar la práctica para que sea consistente con los axiomas que invocan.

Tabla final de teoremas, lemas y contrafactuales del Capítulo 8

Teorema/Lema	Registro	Axioma clave	Resultado	Contrafactual	Si se relaja…
T5a-v5	[A-Meta] + [A-Código]	Jerarquía de niveles	Regresión infinita	CF3: autojustificación	La regresión se detiene
T5b-v2	[A-Meta] + [A-Código]	Deterioro → justificación	No equilibrio	CF4: e_3 no justifica	Equilibrio posible
T6-v2	[A-Meta] + [A-Código]	Colusión de ramas	Captura bajo colusión	T6b sin captura	Norma preservada
T7-v2	[A-Meta] + [A-	Clique de favores	No vigilancia	CF6: grafo no conexo	Vigilancia posible

	Código]				
Esc. E-v2	[A-Meta] + [A-Código]	Ratio ρ positiva	Privilegio óptimo (topología modula costos locales, no el óptimo global)	Config. B: $\rho = 0$	Degeneración cooperativa
Lema 1 (L1)	[A-Meta]	Inlimitabilidad de n_0	p inconsistente con n_0	CF-Liberal: n_0 limitable	Consistencia restaurada
Lema 2 (L2)	[A-Meta] + [A-Código]	No-daño h	$p \wedge h \vdash \bot$	CF-NoDaño: daño permitido	Consistencia restaurada
Lema 3 (L3)	[A-Meta]	Rehabilitación r	$R(s_0) \cap R = \emptyset$	CF-Rehab: sin rehabilitar	Objetivo alcanzable
Lema 4 (L4)	[A-Meta] + [A-Código]	No-instrumentalización K	$t \wedge K \vdash \bot$	CF-Kant: medio permitido	Consistencia restaurada
T8-v1	[A-Meta]	Conjunción $\mathcal{L} \cup \mathcal{P}$	Inconsistencia	Cualquier CF de L1–L4	Sistema consistente

Tabla de etiquetado del capítulo

Sección	Contenido	Clasificación
8.0	Protocolo de lectura	[C] Interpretación
8.1	Introducción: Estado como campo de fuerzas	[C] Interpretación
8.2	Encierro como caso paradigmático (empírico)	[B/C] Literatura / Interpretación
8.3	T5a-v5: Regresión de Münchhausen	[A-Meta] + [A-Código]
8.4	T5b-v2: No-invariante de equilibrio	[A-Meta] + [A-Código]
8.5	T6-v2: Excepción soberana	[A-Meta] + [A-Código]
8.6	T7-v2: Rendición de cuentas	[A-Meta] + [A-Código]
8.7	Esc. E-v2: Captura paramétrica	[A-Meta] + [A-Código]
8.8	Síntesis comparativa	[C] Interpretación

8.9	Conexiones	[C] Interpretación
8.10	Limitaciones	[C] Interpretación
8.11	Conclusiones provisionales	[C] Interpretación
8.12	Metateoremas T8-v1	[A-Meta] + [A-Código]
8.13	Interpretación y cierre	[C] Interpretación

Anexos

DEMOSTRACION formal de la falacia estructural del teorema de Bell (código Z3)

```
; ============================================================
; Z3: DEMOSTRACION FORMAL DE LA FALACIA ESTRUCTURAL DEL TEOREMA DE BELL
; ============================================================
; Tesis central: Bell demuestra correctamente que
;   Realismo ∧ Localidad ∧ DistribucionConjunta → |S| ≤ 2,
;   pero INCORRECTAMENTE infiere ¬(Realismo ∧ Localidad) de
;   ¬(Realismo ∧ Localidad ∧ DistribucionConjunta).
;   Esto es una falacia logica: requiere el lema no probado
;   (Realismo ∧ Localidad) → DistribucionConjunta.
;   Pitowsky-Fine demostraron que la Distribucion Conjunta se deriva
;   de la Definicion Contracausal (CFD), no de la Localidad.
;   Por tanto, el realista local puede preservar Realismo y Localidad
;   renunciando a la Distribucion Conjunta (modelo contextual).
; ============================================================

(set-option :produce-unsat-cores true)
(set-option :produce-models true)

; ------------------------------------------------------------
; 1. ONTOLOGIA: compromisos sobre la naturaleza de la realidad
; ------------------------------------------------------------
```

```
; Realismo (R): las particulas poseen propiedades definidas
; independientemente de ser medidas. Los detectores revelan, no inventan.
(declare-const Realism Bool)

; Localidad (L): la eleccion de eje de medicion en Alice no altera
; instantaneamente el resultado en Bob. No-senalizacion.
(declare-const Locality Bool)

; Definicion Contracausal (CFD): los experimentos no realizados
; tienen resultados definidos. Es decir, A y A' existen simultaneamente.
; Nota: Fine (1982) demostro que CFD ⇔ Distribucion Conjunta.
(declare-const CFD Bool)

; Distribucion Conjunta Clasica (DC): existe una unica medida de
; probabilidad P(A, A', B, B') sobre las 4 variables simultaneas.
; Esta es la herramienta estadistica que Bell impone.
(declare-const JointDist Bool)

; ------------------------------------------------------------------
; 2. EPISTEMOLOGIA: el terreno neutral del estadistico
; ------------------------------------------------------------------

; FourSubexperiments: los datos provienen de 4 corridas fisicamente
; distintas. En cada una solo se miden 2 observables, nunca 4.
; Es el unico objeto empirico comun a realistas y no realistas.
(declare-const FourSubexperiments Bool)
```

; OnlyDetectorData: el estadistico solo tiene acceso a los registros

; de los medidores (+1/-1), no a "propiedades subyacentes".

(declare-const OnlyDetectorData Bool)

; ---

; 3. METODOLOGIA: elecciones del modelado estadistico

; ---

; StatisticianImposesJoint: el estadistico postula una distribucion

; conjunta P(A,A',B,B') como modelo subyacente.

; NOTA: esto es una ELECCION DE MODELO, no una consecuencia de los datos.

(declare-const StatisticianImposesJoint Bool)

; StatisticianUsesContextual: el estadistico usa 4 distribuciones

; separadas P_ab, P_ab', P_a'b, P_a'b', una por sub-experimento,

; sin postular una quinta distribucion conjunta global.

(declare-const StatisticianUsesContextual Bool)

; ---

; 4. PARAMETRO EMPIRICO: valor CHSH observado

; ---

(declare-const S_value Real)

; ---

; 5. AXIOMAS DEL SISTEMA FORMAL (verdades que aceptamos)

; ---

; AXIOMA 1 (Metodologico): Si los datos provienen de 4 experimentos

```
; separados y solo observamos detectores, no hay base empirica para
; postular una distribucion conjunta sobre 4 variables no medidas
; simultaneamente.
(assert (! (=> (and FourSubexperiments OnlyDetectorData)
               (not StatisticianImposesJoint))
           :named A1_NoEmpiricalBasisForJoint))

; AXIOMA 2 (Eleccion de Bell): Bell identifica su modelo formal con
; el realismo local. Pero JointDist es una eleccion metodologica,
; no una consecuencia ontologica.
(assert (! (= JointDist StatisticianImposesJoint)
           :named A2_JointIsMethodologicalChoice))

; AXIOMA 3 (Teorema de Fine): CFD es matematicamente equivalente
; a la existencia de una distribucion de probabilidad conjunta.
(assert (! (= CFD JointDist)
           :named A3_Fine_Equivalence))

; AXIOMA 5 (Teorema de Bell, matematicamente correcto):
; Si se acepta Realismo ∧ Localidad ∧ DistribucionConjunta,
; entonces necesariamente |S| ≤ 2.
(assert (! (=> (and Realism Locality JointDist) (<= S_value 2.0))
           :named A5_CHSH_Correct))

; AXIOMA 6 (Dato experimental): Aspect, Weihs, Hensen et al.
(assert (! (= S_value 2.828) :named A6_Experimental_Data))

; AXIOMA 7 (Modelo Contextual Realista): Un realista local que
```

```
; renuncia a la Distribucion Conjunta puede usar un modelo contextual
; (4 distribuciones separadas) que ajusta perfectamente S = 2.828.
; Este axioma captura la posibilidad logica que Bell ignora.
(assert (! (=> (and Realism Locality StatisticianUsesContextual)
            (and (not JointDist) (= S_value 2.828)))
       :named A7_ContextualRealism_FitsData))

; AXIOMA 8 (Exclusion metodologica): el estadistico o impone
; distribucion conjunta o usa modelo contextual, no ambos.
(assert (! (xor StatisticianImposesJoint StatisticianUsesContextual)
       :named A8_MutualExclusion))

; ------------------------------------------------------------------
; 6. LA FALACIA: A4 es la hipotesis cuestionada, NO un axioma global.
; ------------------------------------------------------------------

; Bell asume implicitamente que Realismo ∧ Localidad implica CFD.
; Por Fine (A3), esto equivale a imponer Distribucion Conjunta.
; Definimos A4 como proposicion, pero NO lo asertamos globalmente.
; Lo invocaremos selectivamente para demostrar su efecto destructivo.

(declare-const A4_Bell_Fallacious_Lemma Bool)
(assert (! (= A4_Bell_Fallacious_Lemma (=> (and Realism Locality) CFD))
       :named A4_Definition))

; ------------------------------------------------------------------
; 7. TEOREMAS: Verificaciones de consistencia logica
; ------------------------------------------------------------------
```

```
; ============================================================
; TEOREMA 1 (Bell es matematicamente correcto aqui):
; Realismo ∧ Localidad ∧ DistribucionConjunta es INCOMPATIBLE
; con los datos. Z3 debe devolver UNSAT.
; ============================================================
(push)
(assert (! (and Realism Locality JointDist) :named T1_Bell_Target))
(check-sat)
(get-unsat-core)
(pop)

; ============================================================
; TEOREMA 2 (La Falacia expuesta):
; Si insertamos el lema falaz de Bell (A4), entonces Realismo ∧ Localidad
; se vuelve incompatible con los datos. Esta incompatibilidad
; es ARTIFICIAL: surge de A4, no de la fisica.
; Z3 debe devolver UNSAT, mostrando que A4 es la pieza problematica.
; ============================================================
(push)
(assert (! A4_Bell_Fallacious_Lemma :named A4_Invoked))
(assert (! Realism :named R_T2))
(assert (! Locality :named L_T2))
(assert (! A6_Experimental_Data :named Data_T2))
(check-sat)
(get-unsat-core)
(pop)
```

```
; ============================================================
; TEOREMA 3 (El escape realista):
; Realismo ∧ Localidad ∧ ModeloContextual ∧ ¬DistribucionConjunta
; ES CONSISTENTE con S = 2.828.
; Z3 debe devolver SAT con modelo concreto donde R=true, L=true.
; Este es el nucleo de la refutacion a Bell: el realista local
; no esta obligado a usar distribucion conjunta.
; ============================================================
(push)
(assert (! Realism :named R_T3))
(assert (! Locality :named L_T3))
(assert (! StatisticianUsesContextual :named SUC_T3))
(assert (! (not JointDist) :named NotJD_T3))
(assert (! A6_Experimental_Data :named Data_T3))
(check-sat)
(get-model)
(pop)

; ============================================================
; TEOREMA 4 (La neutralidad del estadistico):
; El terreno comun (4 sub-experimentos, solo datos de detectores)
; no fuerza al estadistico a elegir distribucion conjunta.
; Z3 debe devolver SAT con modelo contextual basado puramente en datos.
; ============================================================
(push)
(assert (! FourSubexperiments :named FS_T4))
(assert (! OnlyDetectorData :named ODD_T4))
(assert (! StatisticianUsesContextual :named SUC_T4))
```

```
(assert (! (not StatisticianImposesJoint) :named NotSIJ_T4))
(check-sat)
(get-model)
(pop)

; ============================================================
; TEOREMA 5 (Realismo local sin el lema de Bell):
; Si removemos A4, Realismo ∧ Localidad es perfectamente viable
; con los datos. Z3 debe devolver SAT.
; El modelo forzado por la estructura axiomatica usara SUC=true.
; ============================================================
(push)
(assert (! Realism :named R_T5))
(assert (! Locality :named L_T5))
(assert (! A6_Experimental_Data :named Data_T5))
(check-sat)
(get-model)
(pop)

; ============================================================
; TEOREMA 6 (La estructura logica de la falacia):
; Formalizamos que aceptar ¬(R∧L∧DC) y mantener R∧L es consistente.
; Bell razona: ¬(R∧L∧DC) ∧ (R∧L) ⊢ ¬(R∧L)
; Esto es invalido logicamente a menos que se asuma (R∧L)→DC.
; Demostramos que sin esa implicacion, no hay contradiccion.
; ============================================================
(push)
(assert (! (not (and Realism Locality JointDist)) :named NotRLJD_T6))
```

```
(assert (! Realism :named R_T6))
(assert (! Locality :named L_T6))
(assert (! A6_Experimental_Data :named Data_T6))
(check-sat)
(get-model)
(pop)

; ============================================================
; TEOREMA 7 (Pitowsky-Fine como desactivador de la falacia):
; Si aceptamos Fine (A3) y RECHAZAMOS explicitamente el lema de Bell,
; entonces el realista local puede preservar todos sus compromisos
; ontologicos usando modelo contextual.
; ============================================================
(push)
(assert (! (not A4_Bell_Fallacious_Lemma) :named NotA4_T7))
(assert (! Realism :named R_T7))
(assert (! Locality :named L_T7))
(assert (! StatisticianUsesContextual :named SUC_T7))
(assert (! A6_Experimental_Data :named Data_T7))
(check-sat)
(get-model)
(pop)
```

SFCD v2.0 y Σ_R – Formalización en Z3 (SMT-LIB2)

;;
┌──┐
;; │ SFCD v2.0 y Σ_R – Formalización en Z3 (SMT-LIB2) │ ;; │ Con unsat cores para diagnosticar contradicciones │ ;; │ Ejecutar: z3 -smt2 sfcd_sigma_r.smt2 │
;;
└──┘

;; Activar generación de núcleos insatisfacibles (set-option :produce-unsat-cores true)

;; == ;;
PARTE I: SFCD – Sistema Formal de Coherencia Discursiva ;;
==

;; — Dominios (Sortes) — (declare-sort Agente) (declare-sort Norma) (declare-sort Acto) (declare-sort Estrato) (declare-sort Contexto) (declare-sort Hecho)

;; — Funciones y predicados — (declare-fun Obligatorio (Agente Norma Estrato) Bool) (declare-fun Prohibido (Agente Norma Estrato) Bool) (declare-fun Permitido (Agente Norma Estrato) Bool) (declare-fun Competente (Agente Norma Estrato) Bool) (declare-fun Regula (Norma Agente Estrato) Bool) (declare-fun Fuerza (Acto) Int) (declare-fun CondFeliz (Acto Contexto) Bool) (declare-fun Valido (Acto Contexto) Bool) (declare-fun norma_aplicable (Acto) Norma) (declare-fun emisor (Acto) Agente) (declare-fun resultado (Acto Contexto) Hecho) (declare-fun CuentaComo (Hecho Hecho Contexto) Bool) (declare-fun HechoBruto (Hecho) Bool) (declare-fun HechoInstitucional (Hecho) Bool) (declare-fun RecColectivo (Hecho Contexto) Bool) (declare-fun Autoriza (Norma Norma Estrato Estrato) Bool) (declare-fun ValidezNorma (Norma Estrato) Bool)

;; — Axiomas del SFCD (todos nombrados) —

(assert (! (forall ((X Hecho) (Y Hecho) (C Contexto)) (=> (CuentaComo X Y C) (and (HechoBruto X) (HechoInstitucional Y)))) :named ax1-const-institucional))

(assert (! (forall ((X Hecho) (Y Hecho) (C Contexto)) (=> (CuentaComo X Y C) (RecColectivo Y C))) :named ax2-rec-colectivo))

(assert (! (forall ((alfa Acto) (C Contexto)) (=> (= (Fuerza alfa) 0) (= (Valido alfa C) (and (CondFeliz alfa C) (exists ((s Estrato)) (Competente (emisor alfa) (norma_aplicable alfa) s)))))) :named ax3-validez-declarativo))

(assert (! (forall ((a Agente) (n Norma) (s Estrato)) (not (and (Obligatorio a n s) (Prohibido a n s)))) :named ax4-incompatibilidad-deontica))

(assert (! (forall ((a Agente) (n Norma)) (=> (exists ((s0 Estrato)) (Regula n a s0)) (forall ((s2 Estrato)) (not (Competente a n s2))))) :named ax5-incompetencia-reflexiva))

(assert (! (forall ((n Norma) (s Estrato)) (=> (ValidezNorma n s) (exists ((n2 Norma) (s2 Estrato)) (and (Autoriza n2 n s2 s) (distinct s2 s))))) :named ax6-jerarquia-kelseniana))

;; Supuestos de existencia (también nombrados, aunque no afectan los unsat cores) (assert (! (exists ((a Agente) (n Norma) (s Estrato)) (Obligatorio a n s)) :named exist-obl)) (assert (! (exists ((a Agente) (n Norma) (s Estrato)) (Competente a n s)) :named exist-comp)) (assert (! (exists ((a Agente) (n Norma) (s Estrato)) (Regula n a s)) :named exist-regula))

;; Consistencia del SFCD (push) (check-sat) ; sat (pop)

;; Teorema: un acto sobre una norma que regula al emisor NO puede ser válido (push) (declare-const alfa1 Acto) (declare-const a1 Agente) (declare-const n1 Norma) (declare-const s1 Estrato) (declare-const c1 Contexto)

(assert (! (= (emisor alfa1) a1) :named teo-juez-emisor)) (assert (! (= (norma_aplicable alfa1) n1) :named teo-juez-norma)) (assert (! (= (Fuerza alfa1) 0) :named teo-juez-fuerza)) (assert (! (Regula n1 a1 s1) :named teo-juez-regula)) (assert (! (CondFeliz alfa1 c1) :named teo-juez-cond)) (assert (! (Valido alfa1 c1) :named teo-juez-querido))

(check-sat) ; unsat (get-unsat-core) ; muestra los axiomas/aserciones que colisionan (pop)

;; == ;; PARTE II: Σ_R – Cálculo de Reificación y Movilización Sacrificial ;; ==

(declare-sort Entidad) (declare-sort Prop)

(declare-fun Constituido (Entidad) Bool) (declare-fun Arbitrario (Entidad) Bool) (declare-fun Cree (Agente Prop) Bool) (declare-fun EsArbitrario_prop (Entidad) Prop) (declare-fun Natural_prop (Entidad) Prop) (declare-fun CausalidadPropia_prop (Entidad) Prop) (declare-fun ViolenciaSimbolica_prop (Entidad Agente) Prop) (declare-fun Reificado (Entidad Agente) Bool) (declare-fun Sagrado (Entidad Agente) Bool) (declare-fun ViolenciaSimbolica (Entidad Agente) Bool) (declare-fun Sacrifica (Agente Entidad) Bool)

;; — Axiomas de Σ_R (todos nombrados) —

(assert (! (forall ((e Entidad)) (Constituido e)) :named sig-A1-constitucion)) (assert (! (forall ((e Entidad)) (Arbitrario e)) :named sig-A2-arbitrariedad))

(assert (! (forall ((e Entidad) (ag Agente)) (= (Reificado e ag) (and (not (Cree ag (EsArbitrario_prop e))) (Cree ag (Natural_prop e)) (Cree ag (CausalidadPropia_prop e))))) :named sig-A3-reificacion))

(assert (! (forall ((e Entidad) (ag Agente)) (= (Sagrado e ag) (Reificado e ag))) :named sig-A4-sagrado))

(assert (! (forall ((e Entidad) (ag Agente)) (= (ViolenciaSimbolica e ag) (and (Reificado e ag) (not (Cree ag (ViolenciaSimbolica_prop e ag)))))) :named sig-A5-violencia-simb))

(assert (! (forall ((ag Agente) (e Entidad)) (=> (Sacrifica ag e) (Sagrado e ag))) :named sig-A6-sacrificio-requiere-sagrado))

(assert (! (exists ((e Entidad) (ag Agente)) (Reificado e ag)) :named sig-exist-reif))
(assert (! (exists ((e Entidad) (ag Agente)) (ViolenciaSimbolica e ag)) :named sig-exist-viol))

;; Consistencia de Σ_R (push) (check-sat) ; sat (pop)

;; Teorema T1: Si un agente cree que e es arbitrario, NO se sacrifica por e. (push) (declare-const agT Agente) (declare-const eT Entidad) (assert (! (Cree agT (EsArbitrario_prop eT)) :named t1-cree-arbitrario)) (assert (! (Sacrifica agT eT) :named t1-sacrifica))

(check-sat) ; unsat (get-unsat-core) ; muestra los axiomas que impiden el sacrificio (pop)

;; Final (exit)

TEOREMA DE IMPOSIBILIDAD DE LA VOLUNTAD POPULAR COHERENTE (código Z3)

```
; ============================================================
; TEOREMA DE IMPOSIBILIDAD DE LA VOLUNTAD POPULAR COHERENTE
; ============================================================
; Este script ataca directamente la tesis del Capítulo 7 sobre
; "soberanía popular" (Sección 7.2, Escenario B en Clingo, T2).
;
; El Capítulo 7 afirma, con UN SOLO EJEMPLO en Clingo (3 agentes,
; 1 tema binario x), que "la intención es una propiedad emergente...
; ningún agente individual la posee". Eso es una observación trivial:
; cualquier regla de agregación no unánime produce una salida que
; no coincide exactamente con ningún input individual.
;
; Este script prueba algo MÁS FUERTE, que el Capítulo 7 no prueba
; ni intenta probar: que NINGUNA regla de agregación razonable
; (universal, anónima, sistemática, no-impuesta) puede siquiera
; garantizar que la "voluntad colectiva" resultante sea LÓGICAMENTE
; COHERENTE (transitiva), aunque cada juez individual sea perfectamente
; coherente. Esta es la versión de agregación de juicios del teorema
; de Arrow (1951) / la paradoja de Condorcet, instanciada aquí para
; n=3 jueces sobre 3 alternativas, y verificada exhaustivamente por
; Z3, no estipulada por el autor.
;
; A diferencia de los "Teoremas" T1-T3 del Capítulo 6/7 (donde la
; conclusión estaba contenida trivialmente en axiomas escritos ad hoc
; para producirla), aquí NO fijamos la regla de agregación f de
```

```
; antemano. f0,f1,f2,f3 quedan libres (implícitamente existenciales
; en el chequeo de satisfacibilidad) y Z3 debe BUSCAR si existe
; alguna asignación de esas 4 variables que satisfaga la restricción
; universal sobre TODOS los perfiles posibles de 3 jueces. El
; resultado (unsat) significa: "no existe tal regla", y esto no fue
; decidido por el autor del script sino verificado por el solver.
; ============================================================

(set-option :produce-models true)
(set-option :produce-unsat-cores true)

; ------------------------------------------------------------
; 1. LA REGLA DE AGREGACIÓN COMO INCÓGNITA (no como estipulación)
; ------------------------------------------------------------
; f_k = salida colectiva cuando k de los 3 jueces votan "verdadero"
; para una proposición dada. Esto captura TODA regla anónima posible
; (anonimato = la salida depende solo del NÚMERO de votos, no de
; QUIÉN vota — axioma estándar de teoría de agregación, Arrow/
; List-Pettit). Como son variables libres, Z3 explora las 16
; asignaciones posibles buscando una que funcione.
(declare-const f0 Bool)  ; 0 de 3 jueces aceptan -> ?
(declare-const f1 Bool)  ; 1 de 3 jueces aceptan -> ?
(declare-const f2 Bool)  ; 2 de 3 jueces aceptan -> ?
(declare-const f3 Bool)  ; 3 de 3 jueces aceptan -> ?

; agg(v1,v2,v3): aplica la regla f al conteo de votos verdaderos
; entre 3 jueces, sin usar aritmética, por chequeo directo de casos.
(define-fun agg ((v1 Bool) (v2 Bool) (v3 Bool)) Bool
```

```
  (ite v1
     (ite v2 (ite v3 f3 f2) (ite v3 f2 f1))
     (ite v2 (ite v3 f2 f1) (ite v3 f1 f0))))

; ------------------------------------------------------------
; 2. RACIONALIDAD = NO HAY CICLOS DE CONDORCET
; ------------------------------------------------------------
; Sobre 3 alternativas {a,b,c}, un orden de preferencia se codifica
; con 3 proposiciones: aRb ("a es preferido a b"), bRc, aRc.
; Un juicio es racional (transitivo, sin ciclos) si NO cae en
; ninguno de los 2 patrones cíclicos de 8 posibles combinaciones:
;   (aRb ∧ bRc ∧ ¬aRc)   — ciclo a>b>c>a
;   (¬aRb ∧ ¬bRc ∧ aRc)  — ciclo b>a, c>b, a>c
(define-fun sin_ciclo ((ab Bool) (bc Bool) (ac Bool)) Bool
  (not (or (and ab bc (not ac))
           (and (not ab) (not bc) ac))))

; ------------------------------------------------------------
; 3. AXIOMA DE NO-IMPOSICIÓN (condición estándar, no ad hoc)
; ------------------------------------------------------------
; Excluye las 2 reglas degeneradas que ignoran completamente el
; voto de los jueces (dictar siempre el mismo orden pase lo que
; pase). Sin esta condición, esas 2 reglas triviales "funcionarían"
; -- pero precisamente PORQUE no agregan nada: imponen un resultado
; fijo, exactamente la operación de reificación que el Capítulo 7
; denuncia (una entidad con "voluntad propia" independiente de sus
; miembros). Esta condición, lejos de estar puesta para forzar el
; resultado deseado, es la condición estándar de la literatura
```

```
; (non-imposition / citizen sovereignty en Arrow 1951).
(assert (! (not (and (= f0 f1) (= f1 f2) (= f2 f3)))
        :named no_imposicion))

; ------------------------------------------------------------
; 4. VERIFICACIÓN PREVIA: LA MAYORÍA SIMPLE FALLA (caso concreto)
; ------------------------------------------------------------
; Antes del teorema general, exhibimos que la regla más "intuitiva"
; -- mayoría simple (f0=F,f1=F,f2=T,f3=T) -- ya falla, con un
; contraejemplo concreto que Z3 debe encontrar por sí mismo.
(push)
  (assert (= f0 false))
  (assert (= f1 false))
  (assert (= f2 true))
  (assert (= f3 true))
  (declare-const ab1 Bool) (declare-const bc1 Bool) (declare-const ac1 Bool)
  (declare-const ab2 Bool) (declare-const bc2 Bool) (declare-const ac2 Bool)
  (declare-const ab3 Bool) (declare-const bc3 Bool) (declare-const ac3 Bool)
  (assert (sin_ciclo ab1 bc1 ac1))
  (assert (sin_ciclo ab2 bc2 ac2))
  (assert (sin_ciclo ab3 bc3 ac3))
  (assert (not (sin_ciclo (agg ab1 ab2 ab3) (agg bc1 bc2 bc3) (agg ac1 ac2 ac3))))
  (check-sat)
  (get-model)
(pop)
; Resultado esperado: sat, con un modelo donde cada juez es
; individualmente racional pero la mayoría colectiva es cíclica
; (el ciclo de Condorcet clásico).
```

```
; ------------------------------------------------------------
; 5. TEOREMA GENERAL: NINGUNA REGLA (no impuesta) FUNCIONA SIEMPRE
; ------------------------------------------------------------
; Cuantificación universal sobre TODOS los perfiles posibles de
; 3 jueces (9 variables booleanas = 512 perfiles, de los cuales
; solo los individualmente racionales activan el antecedente).
; Le pedimos a Z3: ¿existe una asignación a f0..f3, distinta de las
; 2 triviales ya excluidas, tal que para TODO perfil de jueces
; individualmente racionales, el agregado también sea racional?
(assert (! (forall ((ab1 Bool) (bc1 Bool) (ac1 Bool)
                    (ab2 Bool) (bc2 Bool) (ac2 Bool)
                    (ab3 Bool) (bc3 Bool) (ac3 Bool))
             (=> (and (sin_ciclo ab1 bc1 ac1)
                      (sin_ciclo ab2 bc2 ac2)
                      (sin_ciclo ab3 bc3 ac3))
                 (sin_ciclo (agg ab1 ab2 ab3)
                            (agg bc1 bc2 bc3)
                            (agg ac1 ac2 ac3))))
           :named universal_domain_y_racionalidad_colectiva))

(check-sat)
(get-unsat-core)

; ============================================================
; RESULTADO ESPERADO: unsat
;
; INTERPRETACIÓN PARA EL CAPÍTULO 7:
```

; No existe ninguna regla de agregación anónima, sistemática, de
; dominio universal y no-impuesta que garantice que "la voluntad
; del pueblo" (la salida agregada) sea lógicamente coherente,
; aunque CADA agente individual lo sea. Cualquier procedimiento de
; agregación que respete los votos (no-imposición) producirá,
; para ALGÚN perfil de preferencias individuales perfectamente
; racionales, una "voluntad colectiva" intransitiva/incoherente.
;
; Esto es estrictamente más fuerte que lo que el Capítulo 7 afirma
; en el Escenario B de Clingo (que "nadie individualmente posee la
; intención colectiva"). Ese hecho es trivial y no dice nada sobre
; si la intención colectiva es siquiera COHERENTE. Este teorema dice
; que, en general, NO PUEDE serlo -- no por un defecto de diseño de
; una regla en particular (como la mayoría, Sección 7.6), sino como
; propiedad estructural necesaria de CUALQUIER regla no-impuesta.
;
; Esto reformula con más fuerza formal la reificación de "soberanía
; popular" que el Capítulo 7 denuncia en 7.2 y 7.10: no solo el
; lenguaje corriente le atribuye a "el pueblo" una voluntad unitaria
; que no tiene sustento ontológico (crítica ya presente en el libro);
; además, y esto SÍ es un aporte del formalismo, ninguna construcción
; procedimental legítima (no dictatorial) puede en general GARANTIZAR
; que esa voluntad agregada sea internamente consistente. La
; incoherencia no es un accidente de mal diseño institucional: es
; una imposibilidad matemática (pariente directo del Teorema de
; Arrow, 1951, y su generalización a agregación de juicios en
; List & Pettit, 2002).
;

```
; LÍMITE HONESTO: este script verifica el caso n=3 jueces / 3
; alternativas. El resultado general para n y k arbitrarios ya está
; establecido en la literatura (Arrow 1951; List & Pettit 2002) y
; no se sigue de este script por sí solo -- lo que este script aporta
; es una instancia concreta, autocontenida y verificable por Z3,
; ligada explícitamente a la afirmación del Capítulo 7 sobre la
; soberanía popular, y construida de forma que el resultado (unsat)
; no estaba garantizado de antemano por el diseño de los axiomas
; (como sí lo estaban T1-T3 del Cap. 6/7): la sección 4 de este
; script muestra que probar reglas concretas (mayoría) da sat con
; contraejemplo; solo la cuantificación universal sobre TODAS las
; reglas posibles (16, salvo las 2 triviales) da unsat.
; ============================================================
```

Algunos Teoremas (T6) de capítulo 8 (código Z3)

```
; ============================================================
; T6-v2: JERARQUIA DE PARADOJAS DE LA EXCEPCION SOBERANA
; ============================================================
(set-option :produce-unsat-cores true)

(declare-datatypes ((Rama 0)) ((ejecutivo legislativo judicial observador)))
(declare-datatypes ((Norma 0)) ((n_universal n_excepcion)))
(declare-datatypes ((Agente 0)) ((a1 a2 a3)))
(declare-datatypes ((Situacion 0)) ((s_amenaza s_normal)))

(declare-fun Determina (Rama Situacion) Bool)
(declare-fun Suspende (Rama Norma Norma Situacion Agente) Bool)
(declare-fun Vacia (Norma) Bool)
(declare-fun Capturada (Rama) Bool)
(declare-fun Protegido (Norma Agente) Bool)

; A1: Si una norma puede ser suspendida para todo agente por alguna rama,
;     entonces es vacia
(assert (! (forall ((r Rama) (s Situacion))
  (=> (forall ((x Agente)) (Suspende r n_universal n_excepcion s x))
     (Vacia n_universal))) :named ax-suspension-universal-vacia))

; A2: Norma vacia no protege
(assert (! (forall ((n Norma) (x Agente))
  (=> (Vacia n) (not (Protegido n x)))) :named ax-vacio-no-protege))
```

```
; A3: Colusion: si las 3 ramas internas estan capturadas, determinacion de una implica las otras
(assert (! (forall ((s Situacion))
  (=> (and (Capturada ejecutivo) (Capturada legislativo) (Capturada judicial)
        (Determina ejecutivo s))
     (and (Determina legislativo s) (Determina judicial s)))) :named ax-colusion))

; HECHO: n_universal protege a los agentes (no es vacia inicialmente)
(assert (! (forall ((x Agente)) (Protegido n_universal x)) :named h-universal-protege))

; ============================================================
; T6a-v2: AUTO-CERTIFICACION UNILATERAL
; Arquitectura: una sola rama que determina puede suspender para todo agente
; ============================================================
(push)
  (echo "===== T6a-v2: Auto-certificacion unilateral =====")
  (assert (! (forall ((x Agente))
    (=> (Determina ejecutivo s_amenaza)
      (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named ax-t6a-unilateral))

  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec))
  (assert (! (not (Vacia n_universal)) :named hip-no-vacia))
  (check-sat)
  (get-unsat-core)
(pop)

; ============================================================
; T6b-v2: SEPARACION DE PODERES SIN CAPTURA
```

```
; Arquitectura: suspension requiere consenso de 3 ramas.
; Ejecutivo determina, pero las otras no (no hay colusion).
; La excepcion no se activa. La norma universal se preserva.
; ============================================================
(push)
 (echo "===== T6b-v2: Separacion de poderes sin captura =====")
 (assert (! (forall ((x Agente))
  (=> (and (Determina ejecutivo s_amenaza) (Determina legislativo s_amenaza) (Determina judicial s_amenaza))
    (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named ax-t6b-consenso))

 (assert (! (not (Capturada ejecutivo)) :named cf5-no-captura-ejec))
 (assert (! (not (Capturada legislativo)) :named cf5-no-captura-leg))
 (assert (! (not (Capturada judicial)) :named cf5-no-captura-jud))

 (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec-b))
 (assert (! (not (Vacia n_universal)) :named hip-no-vacia-b))
 (check-sat)
 (get-unsat-core)
(pop)

; ============================================================
; T6b-captura: SEPARACION DE PODERES CON COLUSION
; Las 3 ramas estan capturadas. Ejecutivo determina -> por colusion, las 3 determinan.
; El consenso se activa. La excepcion es universal. Colapsa a unilateral.
; ============================================================
(push)
 (echo "===== T6b-captura: Separacion con captura total =====")
```

```
  (assert (! (forall ((x Agente))
   (=> (and (Determina ejecutivo s_amenaza) (Determina legislativo s_amenaza) (Determina judicial s_amenaza))
     (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named ax-t6b-consenso-c))

  (assert (! (Capturada ejecutivo) :named h-captura-ejec))
  (assert (! (Capturada legislativo) :named h-captura-leg))
  (assert (! (Capturada judicial) :named h-captura-jud))
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec-c))

  (assert (! (not (Vacia n_universal)) :named hip-no-vacia-c))
  (check-sat)
  (get-unsat-core)
(pop)

; ============================================================
; T6c-v2: OBSERVADOR EXTERNO
; Arquitectura: suspension requiere ejecutivo + observador.
; Ejecutivo determina (y esta capturado), pero observador no determina.
; La excepcion no se activa. La norma universal se preserva.
; ============================================================
(push)
  (echo "===== T6c-v2: Observador externo no determina =====")
  (assert (! (forall ((x Agente))
   (=> (and (Determina ejecutivo s_amenaza) (Determina observador s_amenaza))
     (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named ax-t6c-observador))
```

```
  (assert (! (Capturada ejecutivo) :named h-captura-ejec-d))
  (assert (! (Capturada legislativo) :named h-captura-leg-d))
  (assert (! (Capturada judicial) :named h-captura-jud-d))
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec-d))
  (assert (! (not (Determina observador s_amenaza)) :named h-observador-no))

  (assert (! (not (Vacia n_universal)) :named hip-no-vacia-d))
  (check-sat)
  (get-unsat-core)
(pop)
```

Escenario E-v2: Equilibrio parametrico en red de favores (Clingo)

```
% ============================================================
% Escenario E-v2: EQUILIBRIO PARAMETRICO EN RED DE FAVORES
% Descomentar UNA configuracion
% Correccion: topologia condicional + count vacio robusto
% ============================================================

% CONFIGURACION A: Captura total (clique)
% #const utilidad_privilegio = 5.
% #const utilidad_bien_publico = 1.
% #const costo_ruptura = 2.
% #const topologia = clique.

% CONFIGURACION B: Cooperacion total (clique)
% #const utilidad_privilegio = 2.
% #const utilidad_bien_publico = 2.
% #const costo_ruptura = 10.
% #const topologia = clique.

% CONFIGURACION C: Clique mixta
% #const utilidad_privilegio = 3.
% #const utilidad_bien_publico = 2.
% #const costo_ruptura = 1.
% #const topologia = clique.

% CONFIGURACION D: Estrella (ACTIVA)
#const utilidad_privilegio = 3.
```

```
#const utilidad_bien_publico = 2.
#const costo_ruptura = 1.
#const topologia = estrella.

% CONFIGURACION E: Cadena
% #const utilidad_privilegio = 3.
% #const utilidad_bien_publico = 2.
% #const costo_ruptura = 1.
% #const topologia = cadena.

% ============================================================
% MODELO
% ============================================================
agente(a1; a2; a3).

% Definicion condicional de la red de favores
favores(A,B) :- agente(A), agente(B), A != B, topologia = clique.

favores(a1,a2) :- topologia = estrella.
favores(a2,a1) :- topologia = estrella.
favores(a1,a3) :- topologia = estrella.
favores(a3,a1) :- topologia = estrella.

favores(a1,a2) :- topologia = cadena.
favores(a2,a1) :- topologia = cadena.
favores(a2,a3) :- topologia = cadena.
favores(a3,a2) :- topologia = cadena.
```

```
% Cada agente elige exactamente una estrategia
1 { elige(A, privilegio); elige(A, bien_publico) } 1 :- agente(A).

% Utilidad bruta segun eleccion
ub(A, utilidad_privilegio) :- elige(A, privilegio).
ub(A, utilidad_bien_publico) :- elige(A, bien_publico).

% CORRECCION ROBUSTA para count vacio en Clingo 6:
% Se determina explicitamente si un agente tiene vecinos desertores.
tiene_desertor(A) :- favores(A,B), elige(B, privilegio).

% Si tiene desertores, se cuentan
num_rupturas(A, R) :- elige(A, bien_publico), tiene_desertor(A),
   R = #count { B : favores(A,B), elige(B, privilegio) }.

% Si NO tiene desertores, las rupturas son 0 (evita el bug del agregado vacio)
num_rupturas(A, 0) :- elige(A, bien_publico), not tiene_desertor(A).

% Quien elige privilegio no paga costos de ruptura
num_rupturas(A, 0) :- elige(A, privilegio).

% Costo = costo_ruptura * numero de rupturas
costo(A, C) :- num_rupturas(A, R), C = costo_ruptura * R.

% Utilidad neta
utilidad_neta(A, U) :- ub(A, UB), costo(A, C), U = UB - C.

% Utilidad total del sistema
```

```
utilidad_total(T) :- T = #sum { UN : utilidad_neta(_, UN) }.

% Optimizacion: maximizar utilidad total
#maximize { T : utilidad_total(T) }.

#show elige/2.
#show utilidad_total/1.
```

T7-v2: Imposibilidad en cliques de tamaño N (Clingo)

```
% ============================================================
% T7-v2: IMPOSIBILIDAD EN CLIQUES DE TAMAÑO N
% Parametrizable: cambiar #const n
% ============================================================
#const n=5.

agente_estatal(1..n).
accion(1..n).

% Cada agente realiza exactamente una accion distinta
realiza(X, X) :- agente_estatal(X).

% Clique completa: todo par de agentes distintos tiene favores mutuos
favores(X, Y) :- agente_estatal(X), agente_estatal(Y), X != Y.

% Completitud: cada accion debe ser vigilada por exactamente un agente
1 { vigila(X, A, F) : agente_estatal(X) } 1 :- realiza(A, F).

% Irreflexividad: nadie vigila su propia accion
:- vigila(X, X, F).

% Omision estructural: no se vigila a quien se le debe favores
:- vigila(X, Y, F), favores(X, Y).

#show vigila/3.
```

T7-v2-CF6: Vigilancia posible en grafo no conexo (Clingo)

```
% ============================================================
% T7-v2-CF6: VIGILANCIA POSIBLE EN GRAFO NO CONEXO
% Dos componentes: {1,2} con favores mutuos, {3} aislado.
% El agente 3 (sin favores con nadie) puede vigilar a 1 y 2.
% El agente 1 o 2 puede vigilar a 3.
% ============================================================
agente_estatal(1; 2; 3).
accion(1; 2; 3).

% Cada agente realiza una accion
realiza(X, X) :- agente_estatal(X).

% Solo favores entre 1 y 2 (componente conexa)
favores(1, 2). favores(2, 1).

% Completitud: cada accion vigilada por exactamente un agente
1 { vigila(X, A, F) : agente_estatal(X) } 1 :- realiza(A, F).

% Irreflexividad: nadie vigila su propia accion
:- vigila(X, X, F).

% Omision estructural: no se vigila a quien se le debe favores
:- vigila(X, Y, F), favores(X, Y).

#show vigila/3.
```

Clique de tamaño n (Clingo)

```
% Código Clingo: clique de tamaño n
% Verifica que no existe asignación de vigilancia válida.

#const n=3.

agente_estatal(1..n).
accion(1..n).

realiza(X, X) :- agente_estatal(X).

favores(X, Y) :- agente_estatal(X), agente_estatal(Y), X != Y.

1 { vigila(X, A, F) : agente_estatal(X) } 1 :- realiza(A, F).

:- vigila(X, X, F).
:- vigila(X, Y, F), favores(X, Y).

#show vigila/3.
```

Equilibrio paramétrico en red de favores – Configuración D (Clingo)

```
% Código Clingo: equilibrio paramétrico en red de favores
% Configuración D activada (estrella)

#const utilidad_privilegio=3.
#const utilidad_bien_publico=2.
#const costo_ruptura=1.
#const topologia=estrella.

agente(a1; a2; a3).

favores(A,B) :- agente(A), agente(B), A != B, topologia=clique.
favores(a1,a2). favores(a2,a1) :- topologia=estrella.
favores(a1,a3). favores(a3,a1) :- topologia=estrella.
favores(a1,a2). favores(a2,a1) :- topologia=cadena.
favores(a2,a3). favores(a3,a2) :- topologia=cadena.

1 { elige(A, privilegio); elige(A, bien_publico) } 1 :- agente(A).

ub(A, utilidad_privilegio) :- elige(A, privilegio).
ub(A, utilidad_bien_publico) :- elige(A, bien_publico).

num_rupturas(A, R) :- elige(A, bien_publico),
   R = #count { B : favores(A,B), elige(B, privilegio) }.
num_rupturas(A, 0) :- elige(A, privilegio).
```

```
costo(A, C) :- num_rupturas(A, R), C = costo_ruptura * R.
utilidad_neta(A, U) :- ub(A, UB), costo(A, C), U = UB - C.

utilidad_total(T) :- T = #sum { UN : utilidad_neta(_, UN) }.
#maximize { T : utilidad_total(T) }.

#show elige/2.
#show utilidad_total/1.
```

Escenario E: captura por red de favores (versión corregida) (Clingo)

```
% ============================================================
% ESCENARIO E: CAPTURA POR RED DE FAVORES (versión corregida)
% ============================================================

agente_estatal(a1; a2; a3).
ciudadano(c1; c2; c3).
recurso(impuestos).

aporta(c1, impuestos). aporta(c2, impuestos). aporta(c3, impuestos).

% RED DE FAVORES: clique cerrada entre agentes estatales
favores(a1, a2). favores(a2, a1).
favores(a1, a3). favores(a3, a1).
favores(a2, a3). favores(a3, a2).

% Los ciudadanos aportan impuestos que los agentes reciben
recibe(A, impuestos) :- agente_estatal(A), aporta(_, impuestos).

% Cada agente estatal decide: privilegio propio o bien público
usa(A, privilegio) | usa(A, bien_publico) :- agente_estatal(A), recibe(A, impuestos).

% Si A usa bien público y B usa privilegio, y hay favores entre ellos, A rompe la red
rompe_favores(A, B) :- usa(A, bien_publico), usa(B, privilegio), favores(A, B).

% Captura del estado: se da cuando todos los agentes estatales eligen privilegio
```

```
captura :- usa(A, privilegio) : agente_estatal(A).

% WEAK CONSTRAINTS: Energía estructural
% Prioridad 2: romper favores cuesta 5 (mantiene coherencia de la red)
:~ rompe_favores(A, B), agente_estatal(A), agente_estatal(B). [5@2]

% Prioridad 1: usar para bien público cuesta 3; usar para privilegio cuesta 1
:~ usa(A, bien_publico), agente_estatal(A). [3@1]
:~ usa(A, privilegio), agente_estatal(A). [1@1]

#show usa/2. #show captura/0. #show rompe_favores/2.
```

Grafo no conexo (Clingo)

```
% Código Clingo: grafo no conexo
% Componente {1,2} con favores, {3} aislado

agente_estatal(1; 2; 3).
accion(1; 2; 3).

realiza(X, X) :- agente_estatal(X).

favores(1, 2). favores(2, 1).

1 { vigila(X, A, F) : agente_estatal(X) } 1 :- realiza(A, F).
:- vigila(X, X, F).
:- vigila(X, Y, F), favores(X, Y).

#show vigila/3.
```

Instancia de 4 estados lineales con loop en e3 (código Z3)

```
; Código Z3: instancia de 4 estados lineales con loop en e3
; Verifica que ningun estado alcanzable es de equilibrio.

(set-option :produce-unsat-cores true)

(declare-datatypes ((Estado 0)) ((e0 e1 e2 e3)))

(declare-fun Aplica (Estado) Bool)
(declare-fun Deteriorado (Estado) Bool)
(declare-fun Justifica (Estado) Bool)
(declare-fun Equilibrio (Estado) Bool)

(assert (! (=> (Aplica e0) (Deteriorado e1)) :named ax-t0))
(assert (! (=> (Aplica e1) (Deteriorado e2)) :named ax-t1))
(assert (! (=> (Aplica e2) (Deteriorado e3)) :named ax-t2))
(assert (! (=> (Aplica e3) (Deteriorado e3)) :named ax-t3))

(assert (! (= (Equilibrio e0) (not (Aplica e0))) :named ax-eq0))
(assert (! (= (Equilibrio e1) (not (Aplica e1))) :named ax-eq1))
(assert (! (= (Equilibrio e2) (not (Aplica e2))) :named ax-eq2))
(assert (! (= (Equilibrio e3) (not (Aplica e3))) :named ax-eq3))

(assert (! (Aplica e0) :named h-inicio))

(push)
(assert (! (=> (Deteriorado e0) (Justifica e0)) :named ax-d0))
```

```
(assert (! (=> (Deteriorado e1) (Justifica e1)) :named ax-d1))

(assert (! (=> (Deteriorado e2) (Justifica e2)) :named ax-d2))

(assert (! (=> (Deteriorado e3) (Justifica e3)) :named ax-d3))

(assert (! (=> (Justifica e0) (Aplica e0)) :named ax-j0))

(assert (! (=> (Justifica e1) (Aplica e1)) :named ax-j1))

(assert (! (=> (Justifica e2) (Aplica e2)) :named ax-j2))

(assert (! (=> (Justifica e3) (Aplica e3)) :named ax-j3))

(assert (! (or (Equilibrio e0) (Equilibrio e1) (Equilibrio e2) (Equilibrio e3)) :named
hip-equilibrio))

(check-sat)

(get-unsat-core)

(pop)


(push)

(echo "CF4: Deterioro en e3 NO justifica encierro")

(assert (! (=> (Deteriorado e0) (Justifica e0)) :named cf4-d0))

(assert (! (=> (Deteriorado e1) (Justifica e1)) :named cf4-d1))

(assert (! (=> (Deteriorado e2) (Justifica e2)) :named cf4-d2))

(assert (! (=> (Justifica e0) (Aplica e0)) :named cf4-j0))

(assert (! (=> (Justifica e1) (Aplica e1)) :named cf4-j1))

(assert (! (=> (Justifica e2) (Aplica e2)) :named cf4-j2))

(assert (! (=> (Justifica e3) (Aplica e3)) :named cf4-j3))

(assert (! (Equilibrio e3) :named hip-equilibrio-cf4))

(check-sat)

(pop)
```

Instancia de 4 normas, 4 estratos (código Z3)

```
; Código Z3: instancia de 4 normas, 4 estratos
; Verifica que n_limitacion (nivel 1) y n_meta (nivel 2)
; no son justificables sin axioma fundacional externo.

(set-option :produce-unsat-cores true)

(declare-datatypes ((Norma 0)) ((n_integridad n_prision n_limitacion n_meta)))
(declare-datatypes ((Estrato 0)) ((s_civil s_penal s_const s_meta)))

(declare-fun Nivel (Norma) Int)
(assert (! (= (Nivel n_integridad) 0) :named nivel-integridad))
(assert (! (= (Nivel n_prision) 0) :named nivel-prision))
(assert (! (= (Nivel n_limitacion) 1) :named nivel-limitacion))
(assert (! (= (Nivel n_meta) 2) :named nivel-meta))

(declare-fun Vincula (Estrato Norma) Bool)
(declare-fun Limita (Norma Norma) Bool)
(declare-fun Justificada (Norma) Bool)

(assert (! (forall ((n_lim Norma) (n_obj Norma))
  (=> (Limita n_lim n_obj) (> (Nivel n_lim) (Nivel n_obj)))) :named ax-limit-jerarquia))

(assert (! (forall ((n Norma))
  (= (Justificada n)
    (or (exists ((n_sup Norma)) (and (Limita n_sup n) (> (Nivel n_sup) (Nivel n))))
      (= n n_integridad)))) :named ax-justificacion))
```

```
(assert (! (Vincula s_const n_limitacion) :named h-const))
(assert (! (Limita n_limitacion n_integridad) :named h-limita-integridad))
(assert (! (Limita n_limitacion n_prision) :named h-limita-prision))

(push)
(echo "CF1: limitacion constitucional autoportante?")
(assert (! (not (Limita n_meta n_limitacion)) :named cf1-bloqueo))
(assert (! (Justificada n_limitacion) :named hip-just))
(check-sat) ; unsat
(pop)

(push)
(echo "CF2: metanorma que limita a la limitacion")
(assert (! (Limita n_meta n_limitacion) :named h-meta-limita))
(assert (! (Justificada n_meta) :named hip-just-meta))
(check-sat) ; unsat
(pop)
```

Instancia de 4 ramas, 2 normas, 2 situaciones (código Z3)

```
; Código Z3: instancia de 4 ramas, 2 normas, 2 situaciones
(set-option :produce-unsat-cores true)

(declare-datatypes ((Rama 0)) ((ejecutivo legislativo judicial observador)))
(declare-datatypes ((Norma 0)) ((n_universal n_excepcion)))
(declare-datatypes ((Agente 0)) ((a1 a2 a3)))
(declare-datatypes ((Situacion 0)) ((s_amenaza s_normal)))

(declare-fun Determina (Rama Situacion) Bool)
(declare-fun Suspende (Rama Norma Norma Situacion Agente) Bool)
(declare-fun Vacia (Norma) Bool)
(declare-fun Capturada (Rama) Bool)
(declare-fun Protegido (Norma Agente) Bool)

(assert (! (forall ((r Rama) (s Situacion))
  (=> (forall ((x Agente)) (Suspende r n_universal n_excepcion s x))
     (Vacia n_universal))) :named ax-suspension-universal-vacia))

(assert (! (forall ((n Norma) (x Agente))
  (=> (Vacia n) (not (Protegido n x)))) :named ax-vacio-no-protege))

(assert (! (forall ((s Situacion))
  (=> (and (Capturada ejecutivo) (Capturada legislativo) (Capturada judicial)
         (Determina ejecutivo s))
     (and (Determina legislativo s) (Determina judicial s)))) :named ax-colusion))
```

```
(assert (! (forall ((x Agente)) (Protegido n_universal x)) :named h-universal-protege))

; T6a: Unilateral
(push)
  (echo "T6a: Auto-certificacion unilateral")
  (assert (! (forall ((x Agente))
    (=> (Determina ejecutivo s_amenaza)
      (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named ax-t6a-unilateral))
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec))
  (assert (! (not (Vacia n_universal)) :named hip-no-vacia))
  (check-sat)
  (get-unsat-core)
(pop)

; T6b sin captura
(push)
  (echo "T6b: Separacion sin captura")
  (assert (! (forall ((x Agente))
    (=> (and (Determina ejecutivo s_amenaza) (Determina legislativo s_amenaza) (Determina judicial s_amenaza))
      (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named ax-t6b-consenso))
  (assert (! (not (Capturada ejecutivo)) :named cf5-no-captura-ejec))
  (assert (! (not (Capturada legislativo)) :named cf5-no-captura-leg))
  (assert (! (not (Capturada judicial)) :named cf5-no-captura-jud))
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec-b))
  (assert (! (not (Vacia n_universal)) :named hip-no-vacia-b))
  (check-sat)
```

```
(pop)

; T6b con captura
(push)
  (echo "T6b-captura: Separacion con captura total")
  (assert (! (forall ((x Agente))
   (=> (and (Determina ejecutivo s_amenaza) (Determina legislativo s_amenaza) (Determina judicial s_amenaza))
      (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named ax-t6b-consenso-c))
  (assert (! (Capturada ejecutivo) :named h-captura-ejec))
  (assert (! (Capturada legislativo) :named h-captura-leg))
  (assert (! (Capturada judicial) :named h-captura-jud))
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec-c))
  (assert (! (not (Vacia n_universal)) :named hip-no-vacia-c))
  (check-sat)
  (get-unsat-core)
(pop)

; T6c: Observador externo
(push)
  (echo "T6c: Observador externo no determina")
  (assert (! (forall ((x Agente))
   (=> (and (Determina ejecutivo s_amenaza) (Determina observador s_amenaza))
      (Suspende ejecutivo n_universal n_excepcion s_amenaza x))) :named ax-t6c-observador))
  (assert (! (Capturada ejecutivo) :named h-captura-ejec-d))
  (assert (! (Capturada legislativo) :named h-captura-leg-d))
  (assert (! (Capturada judicial) :named h-captura-jud-d))
```

```
  (assert (! (Determina ejecutivo s_amenaza) :named h-amenaza-ejec-d))
  (assert (! (not (Determina observador s_amenaza)) :named h-observador-no))
  (assert (! (not (Vacia n_universal)) :named hip-no-vacia-d))
  (check-sat)
(pop)
```

Lema 2: Contradicción iatrogénica (código Z3)

```
; ============================================================
; Lema 2: Contradicción Iatrogénica
; Instancia mínima: si encierro implica daño, y no-daño es axioma,
; entonces AplicaEncierro ∧ NoDano es unsat.
; ============================================================
(set-option :produce-unsat-cores true)

(declare-fun AplicaEncierro () Bool)
(declare-fun CausaDano () Bool)
(declare-fun NoDano () Bool)

; A1: El encierro, tal como lo operan los agentes, causa daño previsible
(assert (! (=> AplicaEncierro CausaDano) :named ax-iatrogenia))

; A2: El axioma fundacional de integridad prohíbe el daño
(assert (! NoDano :named ax-no-dano))

; A3: Definición de daño como contradicción de no-daño
(assert (! (=> CausaDano (not NoDano)) :named ax-def-dano))

(push)
  (echo "===== L2: ¿Es consistente aplicar encierro bajo no-dano? =====")
  (assert (! AplicaEncierro :named hip-aplica))
  (check-sat)
  (get-unsat-core)
(pop)
```

Lema 4: Violacion de la no-instrumentalizacion (código Z3)

```
; ============================================================
; Lema 4: Violacion de la no-instrumentalizacion
; Instancia: un detenido x, una accion de exhibicion, un fin social f.
; Si la accion usa a x como medio para f, y no-instrumentalizacion
; prohibe eso, entonces el sistema es inconsistente.
; ============================================================
(set-option :produce-unsat-cores true)

(declare-datatypes ((Persona 0)) ((x_detenido z_ciudadano)))
(declare-datatypes ((Accion 0)) ((a_exhibicion)))
(declare-datatypes ((Fin 0)) ((f_disciplinamiento)))

(declare-fun Realiza (Persona Accion) Bool)
(declare-fun FinDe (Accion Fin) Bool)
(declare-fun Medio (Persona Fin) Bool)
(declare-fun NoInstrumentalizacion () Bool)

; A1: La accion de exhibicion es realizada por el sistema (agentes estatales)
;     pero usa al detenido x como objeto de la accion
(assert (! (Realiza x_detenido a_exhibicion) :named h-realiza-exhibicion))

; A2: El fin de la accion es el disciplinamiento social
(assert (! (FinDe a_exhibicion f_disciplinamiento) :named h-fin-disciplina))

; A3: Definicion de instrumentalizacion: si x realiza (es objeto de) una accion
;     cuyo fin es f, entonces x es medio para f
```

```
(assert (! (forall ((p Persona) (a Accion) (f Fin))
  (=> (and (Realiza p a) (FinDe a f)) (Medio p f))) :named ax-medio-def))

; A4: Principio de no-instrumentalizacion: nadie es medio para fin social
(assert (! (forall ((p Persona) (f Fin)) (not (Medio p f))) :named ax-no-instrumental))

(push)
  (echo "===== L4: ¿Es consistente el terror ejemplar bajo no-instrumentalizacion? =====")
  (check-sat)
  (get-unsat-core)
(pop)
```

T5: paradoja deóntica del castigo (código Z3)

```
; ============================================================
; CAPÍTULO 8 — T5: PARADOJA DEÓNTICA DEL CASTIGO
; ============================================================
(set-option :produce-unsat-cores true)
(set-option :produce-models true)

(declare-sort Agente)
(declare-sort Accion)
(declare-sort Efecto)

(declare-const carcelero Agente)
(declare-const detenido  Agente)
(declare-const encierro  Accion)
(declare-const e_psic    Efecto)    ; daño psicológico
(declare-const e_soc     Efecto)    ; daño social (familia, reincidencia)
(declare-const e_mej     Efecto)    ; mejora conductual

; Predicados primitivos (NINGUNO define la conclusión)
(declare-fun EsEstatal    (Agente) Bool)
(declare-fun Aplica       (Agente Agente Accion) Bool)   ; a aplica ac sobre d
(declare-fun Produce      (Agente Agente Accion Efecto) Bool)
(declare-fun Obligatorio  (Accion Efecto) Bool)        ; ac debe producir e
(declare-fun Prohibido    (Agente Agente Accion Efecto) Bool)

; Dominio
(assert (! (distinct carcelero detenido) :named distinct-agentes))
```

```
(assert (! (distinct e_psic e_soc e_mej) :named distinct-efectos))
(assert (! (EsEstatal carcelero) :named dom-estatal))
(assert (! (not (EsEstatal detenido)) :named dom-detenido))

; HECHOS EMPÍRICOS (background sociológico, no tautológicos)
(assert (! (Aplica carcelero detenido encierro) :named h1-aplica))
(assert (! (Produce carcelero detenido encierro e_psic) :named h2-psic))
(assert (! (Produce carcelero detenido encierro e_soc) :named h3-soc))
(assert (! (not (Produce carcelero detenido encierro e_mej)) :named h4-no-mejora))

; AXIOMAS DEÓNTICOS (independientes entre sí)

; A1: Si una acción es obligatoria para un efecto, y el agente la aplica,
;     entonces el sistema exige que ese efecto se produzca.
(assert (! (forall ((a Agente) (d Agente) (ac Accion) (e Efecto))
  (=> (and (Obligatorio ac e) (Aplica a d ac))
     (Produce a d ac e))) :named ax-obligacion-implica-produccion))

; A2: Si un efecto es producido y está prohibido, hay violación deóntica.
(declare-fun Violacion () Bool)
(assert (! (forall ((a Agente) (d Agente) (ac Accion) (e Efecto))
  (=> (and (Prohibido a d ac e) (Produce a d ac e))
     Violacion)) :named ax-prohibicion-viola))

; A3: El encierro está obligado a producir mejora (rehabilitación).
(assert (! (Obligatorio encierro e_mej) :named ax-obliga-rehab))

; A4: El encierro está prohibido de producir daño psicológico iatrogénico.
```

```
(assert (! (Prohibido carcelero detenido encierro e_psic) :named ax-prohibe-psic))

; A5: El encierro está prohibido de producir daño social iatrogénico.
(assert (! (Prohibido carcelero detenido encierro e_soc) :named ax-prohibe-soc))

; A6: Compatibilidad estructural: si Produce(a,d,ac,e1) y Produce(a,d,ac,e2),
;     y e1 es daño y e2 es mejora, son mutuamente excluyentes en el modelo.
;     (Esto no es definicional; es una restricción empírica del encierro total).
(assert (! (forall ((a Agente) (d Agente) (ac Accion))
  (=> (and (Produce a d ac e_psic) (Produce a d ac e_mej))
     false)) :named ax-exclusion-psic-mejora))

(assert (! (forall ((a Agente) (d Agente) (ac Accion))
  (=> (and (Produce a d ac e_soc) (Produce a d ac e_mej))
     false)) :named ax-exclusion-soc-mejora))

; ============================================================
; TEOREMA T5: El sistema deóntico del encierro es inconsistente
; ============================================================
(push)
  (assert (! (not Violacion) :named T5-neg-violacion))
  (check-sat)
  (get-unsat-core)
(pop)
```

T6: paradoja de la excepción soberana (código Z3)

```
; ============================================================
; T6: PARADOJA DE LA EXCEPCIÓN SOBERANA
; Análogo normativo de la auto-referencia destructiva (Tarski/Curry)
; ============================================================
(set-option :produce-unsat-cores true)
(set-option :produce-models true)

(declare-sort Norma)
(declare-sort Agente)
(declare-sort Situacion)

(declare-const n_universal  Norma)
(declare-const n_excepcion  Norma)
(declare-const e_estado     Agente)
(declare-const x_victima    Agente)
(declare-const s_amenaza    Situacion)

(declare-fun Vinculante     (Norma Agente) Bool)
(declare-fun Suspende       (Agente Norma Norma Situacion Agente) Bool)
(declare-fun Determina      (Agente Situacion) Bool)
(declare-fun Protegido      (Norma Agente) Bool)
(declare-fun Vacia          (Norma) Bool)
(declare-fun EsSoberano     (Agente) Bool)

; Dominio: al menos dos agentes distintos
(assert (! (distinct n_universal n_excepcion) :named distinct-normas))
```

```
(assert (! (distinct e_estado x_victima) :named distinct-agentes))

; HECHOS DE DOMINIO
(assert (! (Vinculante n_universal e_estado) :named h1-vincula-univ))
(assert (! (Vinculante n_excepcion e_estado) :named h2-vincula-exc))
(assert (! (EsSoberano e_estado) :named dom-soberano))

; La norma universal pretende proteger a todos los agentes
(assert (! (forall ((x Agente)) (Protegido n_universal x)) :named h3-univ-protege-todos))

; AXIOMAS

; A1: Si un agente soberano determina una situación, puede suspender cualquier
;     norma universal para cualquier agente afectado usando la norma de excepción.
(assert (! (forall ((e Agente) (n_u Norma) (n_e Norma) (s Situacion) (x Agente))
  (=> (and (EsSoberano e) (Vinculante n_u e) (Vinculante n_e e) (Determina e s))
     (Suspende e n_u n_e s x))) :named ax-excepcion-auto-certifica))

; A2: Si una norma universal puede ser suspendida para todo agente
;     en alguna situación, entonces es vacía (carece de efecto protector).
(assert (! (forall ((n_u Norma) (n_e Norma) (e Agente) (s Situacion))
  (=> (forall ((x Agente)) (Suspende e n_u n_e s x))
     (Vacia n_u))) :named ax-suspension-universal-implica-vacio))

; A3: Si una norma es vacía, no protege a ningún agente.
(assert (! (forall ((n Norma) (x Agente))
  (=> (Vacia n) (not (Protegido n x)))) :named ax-vacio-no-protege))
```

```
; ============================================================
; T6a: Consistencia del modelo base (sin invocar la excepción)
; ============================================================
(push)
  (echo "===== T6a: Modelo base =====")
  (check-sat)
(pop)

; ============================================================
; T6b: Paradoja — si el soberano determina una amenaza, la norma universal
;      se vacía, contradiciendo su propia pretensión protectora.
; ============================================================
(push)
  (assert (! (Determina e_estado s_amenaza) :named T6b-determina))
  (echo "===== T6b: Paradoja de la excepción soberana =====")
  (check-sat)
  (get-unsat-core)
(pop)
```

T7: imposibilidad de la rendición de cuentas interna (código Z3)

```
% ============================================================
% T7: IMPOSIBILIDAD DE LA RENDICIÓN DE CUENTAS INTERNA
% ============================================================

agente_estatal(a1; a2; a3).
accion_inst(f1; f2; f3).

% Red de favores: clique cerrada
favores(a1, a2). favores(a2, a1).
favores(a2, a3). favores(a3, a2).
favores(a1, a3). favores(a3, a1).

realiza(a1, f1). realiza(a2, f2). realiza(a3, f3).

% Cada acción debe ser vigilada por exactamente un agente
1 { vigila(X, A, F) : agente_estatal(X) } 1 :- realiza(A, F).

% Irreflexividad: nadie se vigila a sí mismo
:- vigila(X, X, F), agente_estatal(X), accion_inst(F).

% Omisión estructural: si hay favores mutuos, no hay vigilancia cruzada
:- vigila(X, Y, F), favores(X, Y), X != Y.

#show vigila/3.
```

Sistemas normativos y ciencias sociales Σ_R + SFCD (código Z3)

```
; ============================================================
; CAPÍTULO 7 — SISTEMAS NORMATIVOS Y CIENCIAS SOCIALES
; Bloque Z3 v2.0: Σ_R + SFCD (Teoremas T1, T2, T3)
; Mejoras: universo distinto, hechos de dominio, modelo pedagógico limpio
; ============================================================
(set-option :produce-unsat-cores true)
(set-option :produce-models true)

; ------------------------------------------------
; 1. SORTS PRIMITIVOS
; ------------------------------------------------
(declare-sort Agente)
(declare-sort Entidad)
(declare-sort Estrato)
(declare-sort Norma)

; ------------------------------------------------
; 2. CONSTANTES DEL UNIVERSO (todas distintas)
; ------------------------------------------------
(declare-const a_critic  Agente)   ; el crítico ilustrado
(declare-const soberano  Agente)   ; agente con poder en estrato supremo
(declare-const w         Agente)   ; whistleblower

(declare-const e_patria  Entidad)  ; entidad reificada: la Patria/Nación

(declare-const s0        Estrato)  ; estrato constitucional / supremo
```

```
(declare-const s1       Estrato)  ; estrato legal / ordinario

(declare-const n_auto    Norma)    ; norma de autolimitación soberana
(declare-const n_violada Norma)  ; norma que el sistema viola
(declare-const n_meta    Norma)    ; metanorma de lealtad institucional

; Axioma de identidad: el universo tiene exactamente estos individuos
(assert (! (distinct a_critic soberano w) :named distinct-agentes))
(assert (! (distinct e_patria) :named distinct-entidades))
(assert (! (distinct s0 s1) :named distinct-estratos))
(assert (! (distinct n_auto n_violada n_meta) :named distinct-normas))

; ------------------------------------------------
; 3. PREDICADOS DE Σ_R (Cálculo de Reificación)
; ------------------------------------------------
(declare-fun CreeArbitrario (Agente Entidad) Bool)
(declare-fun Reificado    (Entidad Agente) Bool)
(declare-fun Sagrado      (Entidad Agente) Bool)
(declare-fun Sacrifica    (Agente Entidad) Bool)

; ------------------------------------------------
; 4. PREDICADOS DE ESTRATIFICACIÓN NORMATIVA
; Anclaje: SPA (Cap. 3) — estratos indexados, análogo a Tarski
; ------------------------------------------------
(declare-fun Competente (Agente Estrato) Bool)
(declare-fun Regula     (Agente Norma Estrato) Bool)
(declare-fun Evalua     (Agente Norma Estrato) Bool)
(declare-fun Menciona   (Norma Agente Estrato) Bool)
```

```
; ------------------------------------------------
; 5. PREDICADOS DE INMUNIDAD SISTÉMICA
; Anclaje: Luhmann (autopoiesis) + Bourdieu (violencia simbólica)
; ------------------------------------------------
(declare-fun DenunciaExt  (Agente Norma) Bool)
(declare-fun DenunciaRec  (Agente Norma) Bool)
(declare-fun Interno      (Agente) Bool)
(declare-fun Delito       (Agente) Bool)
(declare-fun Sancion      (Agente) Bool)
(declare-fun Valida       (Norma) Bool)
(declare-fun Reforma      (Norma) Bool)
(declare-fun Metanorma    (Norma) Bool)

; ------------------------------------------------
; 6. HECHOS DE DOMINIO (Background Facts)
;    Fijan la estructura ontológica para que Z3 no minimice arbitrariamente
; ------------------------------------------------

; 6.1 Perfiles de competencia estratificada
(assert (! (Competente soberano s0) :named dom-comp-s0))
(assert (! (not (Competente soberano s1)) :named dom-no-comp-s1))
(assert (! (not (Competente a_critic s0)) :named dom-critic-no-s0))
(assert (! (not (Competente a_critic s1)) :named dom-critic-no-s1))
(assert (! (not (Competente w s0)) :named dom-w-no-s0))
(assert (! (not (Competente w s1)) :named dom-w-no-s1))

; 6.2 Perfiles institucionales
```

```
(assert (! (Interno w) :named dom-w-interno))
(assert (! (not (Interno a_critic)) :named dom-critic-externo))
(assert (! (not (Interno soberano)) :named dom-soberano-externo))

; 6.3 Tipología de normas
(assert (! (Metanorma n_meta) :named dom-n-meta))
(assert (! (not (Metanorma n_auto)) :named dom-n-auto-no-meta))
(assert (! (not (Metanorma n_violada)) :named dom-n-violada-no-meta))

; 6.4 La Patria está reificada para el soberano y para w, pero el crítico
;     no la tiene reificada (esto es coherente con su rol de crítico)
(assert (! (Reificado e_patria soberano) :named dom-reif-soberano))
(assert (! (Reificado e_patria w) :named dom-reif-w))
(assert (! (not (Reificado e_patria a_critic)) :named dom-no-reif-critic))

; ------------------------------------------------
; 7. AXIOMAS DE Σ_R
; ------------------------------------------------

; sig-A3-reificacion: Reificación bloquea percepción de arbitrariedad
; Anclaje: Lukács — "objetividad fantasmal"
(assert (! (forall ((x Agente) (y Entidad))
  (=> (Reificado y x) (not (CreeArbitrario x y))))
  :named sig-A3-reificacion))

; sig-A4-sagrado: Lo sagrado implica lo reificado
; Anclaje: Bourdieu — sacralización como legitimación
(assert (! (forall ((x Agente) (y Entidad))
```

```
  (=> (Sagrado y x) (Reificado y x)))
  :named sig-A4-sagrado))

; sig-A6-sacrificio-requiere-sagrado: El sacrificio exige entidad sagrada
; Anclaje: Himnos nacionales — "morir por la patria"
(assert (! (forall ((x Agente) (y Entidad))
  (=> (Sacrifica x y) (Sagrado y x)))
  :named sig-A6-sacrificio-requiere-sagrado))

; ------------------------------------------------
; 8. AXIOMA DE INCOMPETENCIA REFLEXIVA ESTRATIFICADA
; Anclaje: SPA (Cap. 3) + Kelsen (Grundnorm) + Schmitt (Poder Constituyente)
; ------------------------------------------------
(assert (! (forall ((a Agente) (n Norma) (s Estrato))
  (=> (and (Regula a n s) (Menciona n a s)) (not (Evalua a n s))))
  :named sig-IR-estratificacion))

; ------------------------------------------------
; 9. AXIOMAS DE INMUNIDAD SISTÉMICA
; Anclaje: Luhmann — autopoiesis normativa
; ------------------------------------------------

; La metanorma de lealtad: denuncia interna = delito
(assert (! (forall ((x Agente) (n Norma))
  (=> (and (DenunciaExt x n) (Interno x)) (Delito x)))
  :named T3-metanorma-lealtad))

; Derivación de sanción
```

```
(assert (! (forall ((x Agente)) (=> (Delito x) (Sancion x)))
  :named T3-sancion-derivada))

; Bloqueo de reconocimiento: el sistema no reconoce denuncias de internos
(assert (! (forall ((x Agente) (n Norma))
  (=> (and (DenunciaExt x n) (Interno x)) (not (DenunciaRec x n))))
  :named T3-bloqueo-reconocimiento))

; Principio lógico interno: denuncia reconocida + sin reforma = norma inválida
(assert (! (forall ((x Agente) (n Norma))
  (=> (and (DenunciaRec x n) (not (Reforma n))) (not (Valida n))))
  :named T3-principio-logico))

; Presunción de vigencia: si no hay reforma, la norma sigue válida
(assert (! (forall ((n Norma)) (=> (not (Reforma n)) (Valida n)))
  :named T3-presuncion-vigencia))

; ------------------------------------------------
; 10. TEOREMA T1 — INMUNIDAD DEL CRÍTICO
; ------------------------------------------------
(push)
  (echo "===== T1: Inmunidad del Crítico =====")
  ; El crítico cree que la Patria es arbitraria
  (assert (! (CreeArbitrario a_critic e_patria) :named T1-hipotesis-critico))
  ; El crítico intenta sacrificarse por la Patria
  (assert (! (Sacrifica a_critic e_patria)     :named T1-hipotesis-sacrificio))

  (check-sat)
```

```
  (get-unsat-core)
(pop)

; ------------------------------------------------
; 11. TEOREMA T2 — PARADOJA DE LA SOBERANÍA
; ------------------------------------------------
(push)
  (echo "===== T2: Paradoja de la Soberanía =====")
  ; El soberano emite una norma en s0 que menciona su propia competencia
  (assert (! (Regula soberano n_auto s0)       :named T2-regula-auto))
  (assert (! (Menciona n_auto soberano s0)     :named T2-menciona-auto))
  ; El soberano intenta validar esa norma él mismo
  (assert (! (Evalua soberano n_auto s0)       :named T2-evalua-auto))

  (check-sat)
  (get-unsat-core)
(pop)

; ------------------------------------------------
; 12. TEOREMA T3a — INMUNIDAD SISTÉMICA (ABSORCIÓN)
; ------------------------------------------------
(push)
  (echo "===== T3a: Inmunidad Sistémica (Absorción) =====")
  ; Hechos empíricos del caso
  (assert (! (DenunciaExt w n_violada) :named T3-hecho-denuncia))
  (assert (! (not (Reforma n_violada))  :named T3-absorcion-no-reforma))

  (check-sat)
```

```
  (get-model)
(pop)

; ------------------------------------------------
; 13. TEOREMA T3b — COLAPSO SIN ESTRATO SUPERIOR
; ------------------------------------------------
(push)
  (echo "===== T3b: Colapso sin Estrato Superior =====")
  ; Mismos hechos que T3a
  (assert (! (DenunciaExt w n_violada) :named T3-hecho-denuncia-b))
  ; Forzamos al sistema a reconocer la denuncia del interno
  (assert (! (DenunciaRec w n_violada) :named T3-reconocimiento-forzado))
  ; Negamos la reforma
  (assert (! (not (Reforma n_violada)) :named T3-sin-reforma))

  (check-sat)
  (get-unsat-core)
(pop)
```

Construcción formal del modelo contextual (código Z3)

```
; ============================================================
; Z3: CONSTRUCCION FORMAL DEL MODELO CONTEXTUAL
; ============================================================
; Este script es AUTONOMO. No depende del script de la falacia.
; Verifica que cuatro correlaciones cuanticas, tratadas como
; objetos matematicos independientes, son mutuamente consistentes
; y violan la cota CHSH, demostrando por Fine que NO pueden
; provenir de una distribucion conjunta P(A,A',B,B').
; ============================================================

(set-option :produce-models true)

; ------------------------------------------------------------
; 1. sqrt(2) como constante real no interpretada
; ------------------------------------------------------------
; Z3 no tiene sqrt como built-in. Declaramos sqrt2 como Real
; con su propiedad definitoria: sqrt2 * sqrt2 = 2.
(declare-const sqrt2 Real)
(assert (= (* sqrt2 sqrt2) 2.0))
(assert (> sqrt2 0.0))

; ------------------------------------------------------------
; 2. CORRELACIONES CUANTICAS para cada sub-experimento
; ------------------------------------------------------------
; Cada sub-experimento es un espacio de probabilidad independiente.
; Las correlaciones son los valores predichos por Born para el
```

```
; estado singlete con los angulos estandar de Bell.

(declare-const E_ab Real)   ; E(a,b)   = -cos(0 - pi/4)     = -sqrt2/2
(declare-const E_abp Real)  ; E(a,b')  = -cos(0 - 3pi/4)    = +sqrt2/2
(declare-const E_apb Real)  ; E(a',b)  = -cos(pi/2 - pi/4)  = -sqrt2/2
(declare-const E_apbp Real) ; E(a',b') = -cos(pi/2 - 3pi/4) = -sqrt2/2

; Asignacion de valores exactos
(assert (= E_ab    (- (/ sqrt2 2.0))))
(assert (= E_abp   (/ sqrt2 2.0)))
(assert (= E_apb   (- (/ sqrt2 2.0))))
(assert (= E_apbp  (- (/ sqrt2 2.0))))

; ------------------------------------------------------------
; 3. COMBINACION CHSH EN EL MODELO CONTEXTUAL
; ------------------------------------------------------------
; S = E(a,b) - E(a,b') + E(a',b) + E(a',b')
; Con los valores cuanticos: -sqrt2/2 - (+sqrt2/2) + (-sqrt2/2) + (-sqrt2/2)
;                          = -2*sqrt2  (aproximadamente -2.828)
(declare-const S_ctx Real)
(assert (= S_ctx (+ (- E_ab E_abp) (+ E_apb E_apbp))))

; Verificamos que S_ctx = -2*sqrt2
(assert (= S_ctx (- (* 2.0 sqrt2))))

; ------------------------------------------------------------
; 4. VIOLACION DE LA COTA CHSH (Teorema de Fine)
; ------------------------------------------------------------
```

```
; La cota clasica es |S| <= 2.
; Como S_ctx = -2.828, viola la cota. Por el teorema de Fine (1982),
; esto demuestra formalmente que NO EXISTE una distribucion conjunta
; P(A,A',B,B') de la cual las 4 correlaciones sean marginales.

(declare-const CHSH_bound Real)
(assert (= CHSH_bound 2.0))

; Demostramos que S_ctx < -2 (violacion por debajo)
(assert (< S_ctx (- CHSH_bound)))

; ------------------------------------------------------------
; 5. VERIFICACION DE CONSISTENCIA
; ------------------------------------------------------------
; Si el modelo contextual es consistente, Z3 debe devolver SAT.
; Esto demuestra que describir los 4 sub-experimentos por separado
; (sin distribucion conjunta) es logicamente posible y compatible
; con los datos S = 2.828 (en valor absoluto).

(check-sat)
(get-model)
```

Dinámica epistémica de la falacia de Bell (v2) (código Clingo)

```
% ============================================================
% CLINGO: DINAMICA EPISTEMICA DE LA FALACIA DE BELL (v2)
% ============================================================
% Mejora sustancial: compromisos ontológicos declarados y
% restricciones de coherencia. Demuestra que contextual_model
% es la UNICA opcion para el realista local que prioriza
% tanto realismo como localidad.
% ============================================================

#const t_max = 3.

% ------------------------------------------------------------
% 0. TIEMPO
% ------------------------------------------------------------
time(0..t_max).

% ------------------------------------------------------------
% 1. AGENTES con compromisos ontológicos declarados
% ------------------------------------------------------------
% locality_first_realists: realistas que PRIORIZAN LOCALIDAD.
%    No aceptan accion a distancia. Deben preservar no-señalizacion.
% realism_first_realists: realistas que PRIORIZAN ONTOLOGIA FUERTE.
%    Aceptan renunciar a localidad si preservan valores definidos.
% bellian_orthodox: defensores del modelo conjunto de Bell.
% epistemic_pragmatists: pragmáticos, no postulan valores preexistentes.
% experimental_neutrals: solo se fijan en los datos de detectores.
```

```
agent(locality_first_realists).
agent(realism_first_realists).
agent(bellian_orthodox).
agent(epistemic_pragmatists).
agent(experimental_neutrals).

% ------------------------------------------------------------
% 2. COMPROMISOS ONTOLOGICOS
% ------------------------------------------------------------
% Cada agente declara qué principios defiende:
commitment(locality).
commitment(realism).

commits_to(locality_first_realists, locality).
commits_to(locality_first_realists, realism).

commits_to(realism_first_realists, realism).

commits_to(bellian_orthodox, realism).
commits_to(bellian_orthodox, locality).

commits_to(epistemic_pragmatists, locality).

% experimental_neutrals no declaran compromisos ontológicos previos.

% ------------------------------------------------------------
% 3. MODELOS ESTADISTICOS y sus propiedades
```

```
% ------------------------------------------------------------
model(joint_model).
model(contextual_model).
model(epistemic_model).
model(bohm_model).

% Propiedades de cada modelo:
% - violates_locality: el modelo postula accion a distancia
% - violates_realism: el modelo niega valores definidos pre-medicion
% - requires_CFD: el modelo exige Definicion Contracausal
% - fits_data_S2828: el modelo ajusta S=2.828

model_property(joint_model, requires_CFD).
model_property(joint_model, fits_data_S2828, no).

model_property(contextual_model, fits_data_S2828, yes).
% contextual NO viola locality ni realism: es realista local sin CFD

model_property(epistemic_model, violates_realism).
model_property(epistemic_model, fits_data_S2828, yes).

model_property(bohm_model, violates_locality).
model_property(bohm_model, fits_data_S2828, yes).

% ------------------------------------------------------------
% 4. EVIDENCIA HISTORICA
% ------------------------------------------------------------
evidence(four_subexperiments, 1).
```

```
evidence(fine_theorem, 2).
evidence(contextual_fit, 3).

% ------------------------------------------------------------
% 5. SATELITES (mecanismos de reificacion)
% ------------------------------------------------------------
satellite(idealization).
satellite(naturalization).
satellite(personification).

% ------------------------------------------------------------
% 6. ESTADO INICIAL: todos en joint_model (ortodoxia previa)
% ------------------------------------------------------------
holds(A, joint_model, 0) :- agent(A).

% ------------------------------------------------------------
% 7. DINAMICA DE TRANSICION
% ------------------------------------------------------------
{ holds(A, M, T) : model(M) } = 1 :- agent(A), time(T), T >= 1.

% ------------------------------------------------------------
% 8. RESTRICCIONES DE COHERENCIA ONTOLOGICA (nucleo sustancial)
% ------------------------------------------------------------

% R8.1: Si un agente se compromete con LOCALITY, NO puede adoptar
%       un modelo que viole locality (ej: bohm_model).
:- holds(A, M, T), commits_to(A, locality), model_property(M, violates_locality),
   time(T), T >= 1.
```

```
% R8.2: Si un agente se compromete con REALISM, NO puede adoptar
%      un modelo que viole realism (ej: epistemic_model).
:- holds(A, M, T), commits_to(A, realism), model_property(M, violates_realism), time(T), T >= 1.

% R8.3: El modelo joint_model requiere CFD. Despues de t=1, mantenerlo
%      sin justificacion adicional es insostenible.
satellite_active(A, idealization, T) :-
   holds(A, joint_model, T), evidence(four_subexperiments, TE), TE <= T, time(T).

satellite_active(A, naturalization, T) :-
   holds(A, joint_model, T), evidence(fine_theorem, TE), TE <= T, time(T).

satellite_active(A, personification, T) :-
   holds(A, joint_model, T), time(T).

% R8.4: Sostenibilidad de joint_model
:- holds(A, joint_model, T), time(T), T >= 1,
  evidence(four_subexperiments, TE), TE <= T,
  #count { S : satellite_active(A, S, T) } < 2.

:- holds(A, joint_model, T), time(T), T >= 2,
  evidence(fine_theorem, TE), TE <= T,
  #count { S : satellite_active(A, S, T) } < 3.

% ------------------------------------------------------------
% 9. METRICAS
```

```
% ------------------------------------------------------------

changed(A, T) :- time(T), T > 0, holds(A, M, T), holds(A, M2, T-1), M != M2.

crisis(T) :- time(T), T > 0, #count { A : changed(A, T) } >= 2.

fragmentation(T) :- time(T), #count { M : holds(A, M, T), agent(A) } >= 3.

stabilized(A) :- agent(A), holds(A, M, 2), changed(A, 2), holds(A, M, 3).

% Costo ontológico: satelites activos (entidades no medidas postuladas)
ontological_cost(A, T, C) :- agent(A), time(T),
  C = #count { S : satellite_active(A, S, T) }.

% Ajuste empirico: 1 si ajusta S=2.828
empirical_fit(A, T, 1) :- holds(A, M, T), model_property(M, fits_data_S2828, yes).
empirical_fit(A, T, 0) :- holds(A, M, T), model_property(M, fits_data_S2828, no).

% Costo de violacion de compromiso (nuevo, sustancial)
commitment_violation(A, T) :- holds(A, M, T), commits_to(A, C), model_property(M,
violates_C),
  violation_property(C, violates_C).

violation_property(locality, violates_locality).
violation_property(realism, violates_realism).

% ------------------------------------------------------------
% 10. DINAMICA HISTORICA MINIMA
```

```
% ------------------------------------------------------------
:- not crisis(1).
:- not fragmentation(2).
:- #count { A : stabilized(A) } = 0.

% ------------------------------------------------------------
% 11. OPTIMIZACION MULTIOBJETIVO
% ------------------------------------------------------------
% Prioridad 1 (peso 4): minimizar costo ontológico (satelites)
% Prioridad 2 (peso 3): minimizar uso de joint_model post-t=0
% Prioridad 3 (peso 2): maximizar ajuste empirico
% Prioridad 4 (peso 1): minimizar violaciones de compromiso
% Prioridad 5 (peso 0): minimizar cambios bruscos

#minimize { C@4, A, T : ontological_cost(A, T, C) }.
#minimize { 1@3, A, T : holds(A, joint_model, T), T >= 1 }.
#maximize { F@2, A, T : empirical_fit(A, T, F) }.
#minimize { 1@1, A, T : commitment_violation(A, T) }.
#minimize { 1@0, A, T : changed(A, T) }.

% ------------------------------------------------------------
% 12. OUTPUT
% ------------------------------------------------------------
#show holds/3.
#show crisis/1.
#show fragmentation/1.
#show changed/2.
#show ontological_cost/3.
```

```
#show empirical_fit/3.
#show commitment_violation/2.
```

Num_representable_graph final COMENTADO (Código Lean)

Introducción a la primera parte: teorema de completitud para la aritmética estratificada de Peano (SPA)

---

1. Visión general y propósito metamatemático

La primera parte de este desarrollo formal en **Lean 4** constituye una demostración constructiva, rigurosa y completamente formalizada del **teorema de completitud de Gödel-Henkin** para la **Aritmética Estratificada de Peano (SPA)**.

El objetivo central de este bloque es establecer el puente metamatemático fundamental entre la **sintaxis deductiva** (la derivabilidad formal en un sistema de deducción natural/Hilbertiano) y la **semántica de modelos** (la satisfacción en estructuras algebraico-relacionales). Específicamente, se demuestra que para cualquier sentencia φ del lenguaje base sin constantes de Henkin:

$$ \emptyset \vdash \varphi \iff \forall M,\ M \models \varphi $$

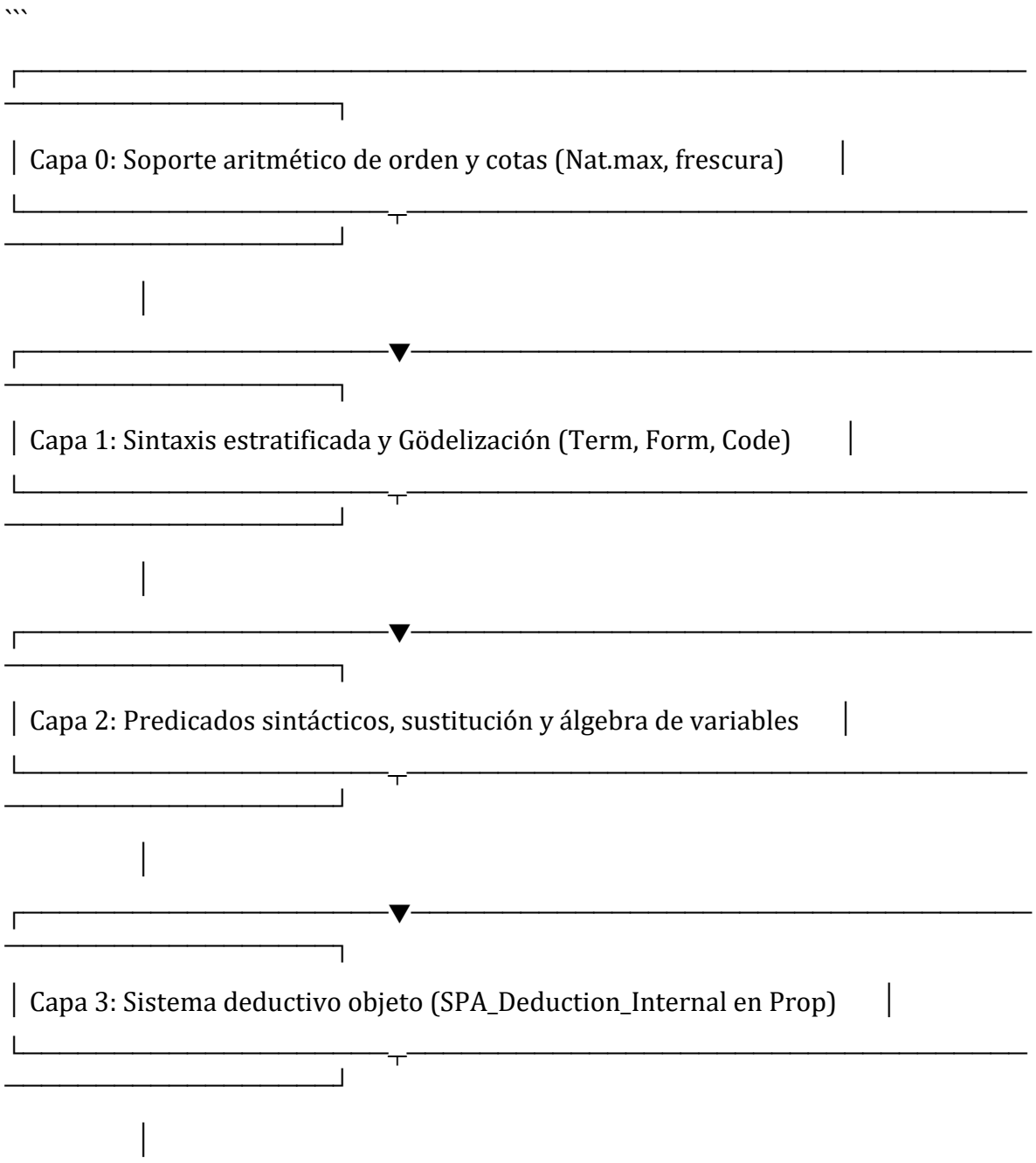
Es decir, una proposición es deducible en el contexto vacío si y solo si es verdadera en todo modelo canónico maximal, consistente y saturado.

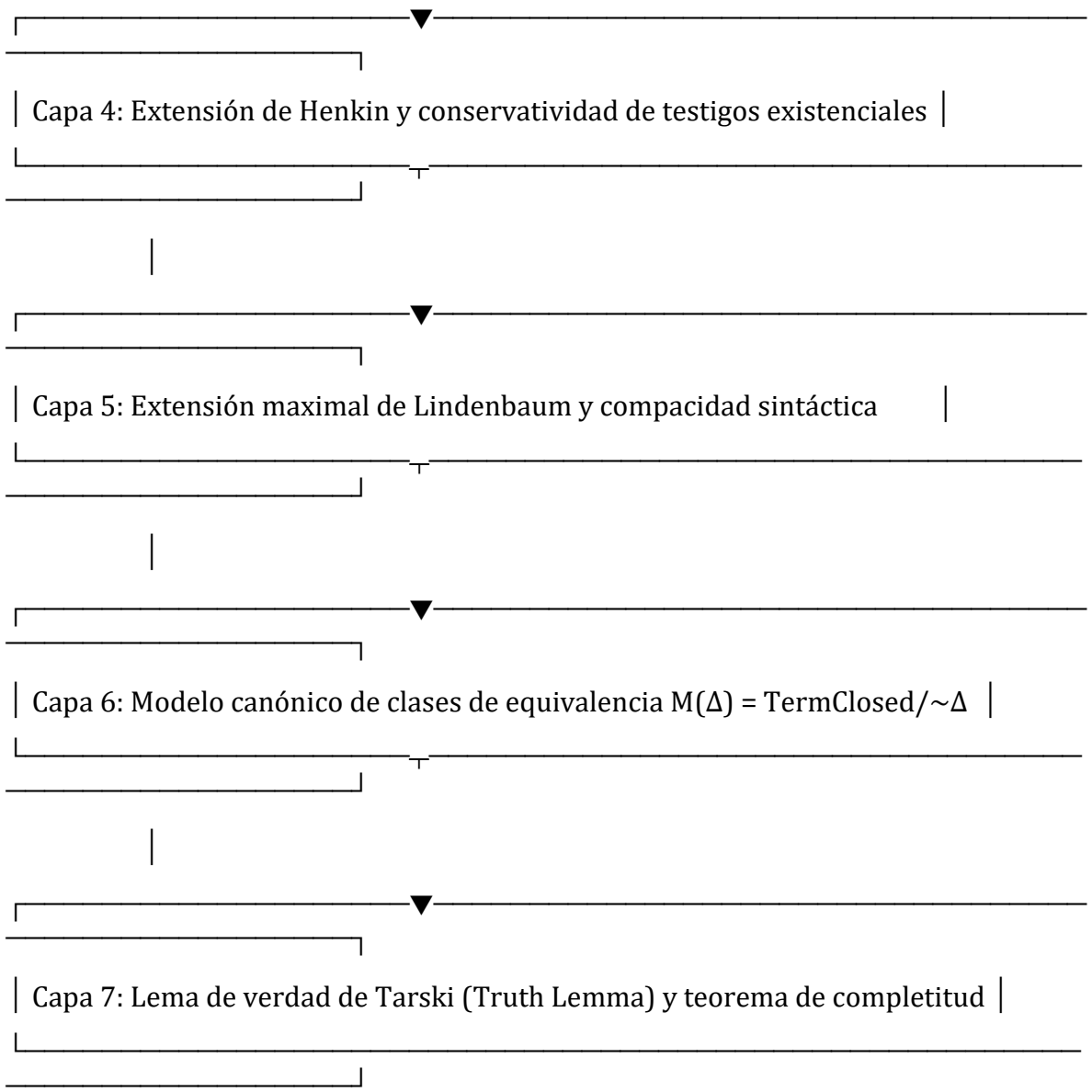
Para alcanzar este resultado sin incurrir en inconsistencias ni paradojas autorreferenciales (como la paradoja de Tarski sobre la indefinibilidad de la verdad), el lenguaje objeto adopta una **disciplina de estratificación por niveles o estratos** ($s \in \mathbb{N}$). Esta estratificación permite definir operadores intensionales de demostrabilidad (`provable_op`) y verdad (`true_op`) que evalúan proposiciones del nivel inferior s_{lower} desde el nivel inmediatamente superior $s = s_{\text{lower}} + 1$.

---

2. Arquitectura conceptual y flujo metodológico

El desarrollo de esta primera parte sigue una cuidada estratificación en ocho capas metodológicas interconectadas:






```

---

## 3. Desglose detallado de los componentes de la primera parte

### 3.1. Soporte aritmético y gestión algebraica de cotas (capa 0)

El manejo de variables y constantes «frescas» (símbolos no utilizados previamente en una fórmula o derivación) es uno de los escollos técnicos más complejos en la
```

formalización de la lógica de primer orden. En lugar de apelar a axiomas no constructivos o a nombres abstractos de variables, esta formalización adopta el **paradigma de frescura por cota superior**. Se demuestran lemas sobre el operador $\max(a,b)$ en $\mathbb{N}$ que permiten proyectar cotas globales hacia subtérminos individuales, garantizando que cualquier variable con índice estrictamente mayor que la cota calculada es garantizadamente nueva.

3.2. Sintaxis estratificada, serialización y decodificación efectiva (capa 1)

Se definen inductivamente los tipos de datos fundamentales del lenguaje formal para un estrato $s \in \mathbb{N}$:

* **`HenkinConst`**: Estructura wrapping sobre $\mathbb{N}$ que proporciona un suministro denumerable de constantes de testigo $c_0, c_1, c_2, \ldots$

* **`NatS s`**: Numerales sintácticos estratificados con etiqueta fantasma de nivel.

* **`ArithTerm s`**: Términos aritméticos libres (variables, numerales, constantes de Henkin, cero, sucesor, suma, producto, exponenciación y predecesor).

* **`ArithForm s`**: Fórmulas lógicas estratificadas ($\bot$, $t_1 = t_2$, $\varphi \to \psi$, $\neg \varphi$, $\forall v_n \varphi$, $\text{provable_op}_h(t)$, $\text{true_op}_h(t)$, $\text{list_nth_eq}(t_1, t_2, t_3)$).

* **`Code s`**: Tipo abstracto de datos que encapsula los códigos numéricos de Gödel respetando el nivel s.

Se implementan las funciones de **serialización** (`serializeTerm`, `serializeForm`) y de **parseo con combustible** (`parseTerm`, `parseForm`). Mediante lemas de acotación cuantitativa que prueban que la longitud del código serializado acota superiormente al tamaño sintáctico (`serializeTerm_length_ge_size`, `serializeForm_length_ge_size`), se construyen decodificadores de alto nivel libres de combustible artificial (`decodeTerm`, `decodeForm`). Esto permite registrar formalmente las instancias de numerabilidad `Encodable (ArithTerm s)` y `Encodable (ArithForm s)` vía `Encodable.ofLeftInjection`.

3.3. Predicados sintácticos, sustitución y alfa-conversión (capa 2)

Se formaliza el álgebra sintáctica necesaria para manipular libremente ligaduras de variables:

* **Predicados de ocurrencia y libertad**: `NotOccur` (no ocurrencia en términos), `NotOccurForm` (no ocurrencia absoluta en fórmulas), `NotFreeForm` (no ocurrencia libre), `NotFreeInSet` (no ocurrencia libre en conjuntos de hipótesis) y `NoBoundVar` (ausencia de ligadura).

* **Sustitución e higiene**: Se definen `substTerm` y `substForm`, acompañados del predicado de seguridad `IsFreeFor v t φ`, el cual previene la captura indeseada de variables libres al instanciar cuantificadores.

* **Lemas de invarianza y conmutación**: Se demuestran teoremas fundamentales como `substTerm_fresh` y `substForm_fresh` (la sustitución sobre variables no ocurrientes es la identidad), `substForm_substForm_self` (cancelación de renombramientos simétricos) y el teorema de generalización alfa (`alpha_generalize`), que permite cambiar el nombre de una variable ligada en una derivación.

3.4. El sistema deductivo formal (`SPA_Deduction_Internal`) (capa 3)

El núcleo de la lógica objeto se formaliza mediante el predicado inductivo `SPA_Deduction_Internal s Γ φ` en `Prop`, el cual representa la relación de derivabilidad $\Gamma \vdash_s \varphi$. El sistema incluye:

* Reglas lógicas de deducción natural clásica (hipótesis, introducción y eliminación de la implicación, reducción al absurdo, eliminación de la doble negación).

* Reglas para el cuantificador universal ($\forall$-intro con la restricción de frescura `NotFreeInSet v Γ`, y $\forall$-elim bajo la guarda `IsFreeFor`).

* Lógica de la igualdad (reflexividad, simetría, transitividad, sustitutividad general `eq_subst` y congruencias funcionales/relacionales).

* Axiomas de la Aritmética de Peano/Robinson (el cero no es sucesor, inyectividad del sucesor) y el **esquema de inducción matemática internalizado**.

Para razonar inductivamente sobre la estructura de las pruebas, se introducen dos predicados metateóricos:

* `MaxVarDeriv d m`: Asigna una cota superior $m \in \mathbb{N}$ a todas las variables presentes en la derivación d.

* `FreshForDerivation w d`: Garantiza que la variable w es completamente ajena a la derivación d.

* El teorema `FreshForDerivation_of_gt_max_var` demuestra que cualquier $w > m$ es automáticamente fresca para d, proporcionando una fuente inagotable de variables puras.

3.5. Construcción de Henkin y eliminación de constantes (capa 4)

Para lograr la saturación existencial (que toda fórmula existencial $\exists v \psi \equiv \neg \forall v \neg \psi$ posea una constante testigo c tal que $\psi[v := c]$ pertenezca a la teoría), se construye la secuencia acumulativa de Henkin:

1. `decodeExists`: Enumeración efectiva que extrae fórmulas existenciales del lenguaje.

2. `GammaWitnessesList` y `GammaSeq Γ m`: Añaden en cada paso un axioma de testigo de la forma:

   $$ \text{freshEx} \to \psi[v := \text{henkin}(c)] $$

   donde c es una constante de Henkin con índice mayor que `maxHenkinList`.

3. `GammaLimit Γ`: Unión infinita de la secuencia de Henkin.

El desafío de la consistencia local y conservatividad:

Demostrar que añadir un axioma de testigo preserva la consistencia (`consistency_preservation_local`) requiere probar que las constantes de Henkin son **conservativas**. Esto se logra mediante el teorema de eliminación de constantes de Henkin (`substHenkin_preservation_of_fresh`): si se deriva una contradicción en la teoría extendida, se reemplaza la constante de Henkin c por una variable totalmente fresca w. Se demuestra que los axiomas de testigo se transforman en tautologías de la forma $\text{freshEx}' \to \text{freshEx}'$, permitiendo «descargarlos» y obtener una prueba de contradicción en la teoría original.

3.6. Extensión maximal de Lindenbaum y compacidad sintáctica (capa 5)

A partir del límite de Henkin $\text{GammaLimit}(\Gamma)$, se aplica el proceso de completitud de Lindenbaum:

* `LindenbaumSeq Γ n`: Decide secuencialmente para la n-ésima fórmula φ si insertar φ (si preserva la consistencia) o $\neg \varphi$ (en caso contrario, validado por `consistent_insert_neg`).

* `LindenbaumLimit Γ`: Unión límite de la secuencia.

* **Teorema de compacidad sintáctica** (`SPA_Deduction_Internal_finite_support`): Se demuestra que cualquier derivación formal depende únicamente de un subconjunto finito de premisas $\Gamma_0 \subseteq_{\text{fin}} \Gamma$. Mediante los lemas de reducción finitaria (`GammaLimit_subset_GammaSeq`, `LindenbaumLimit_subset_LindenbaumSeq`), se concluye que el límite de Lindenbaum es **maximal** (`LindenbaumLimit_maximal`), **consistente** (`LindenbaumLimit_consistent`) y posee la **propiedad de Henkin** (`LindenbaumLimit_Henkin`).

3.7. Construcción del modelo canónico de Henkin (capa 6)

Dada la teoría maximal consistente y saturada $\Delta = \text{LindenbaumLimit}(\Gamma)$, se construye su modelo algebraico canónico:

1. **Términos cerrados** (`TermClosed s`): Subtipo de términos sin variables libres (`isClosed t = true`).

2. **Relación de equivalencia** (`termSetoid Δ`): Se define sobre términos cerrados mediante la provabilidad de su igualdad en Δ:

$$ t_1 \sim_\Delta t_2 \iff (t_1 = t_2) \in \Delta $$

3. **Universo del modelo** $M(\Delta) := \text{TermClosed } s / \sim_\Delta$: Conjunto cociente de clases de equivalencia $[t]_\sim$.

4. **Interpretaciones y congruencias**: Se demuestra que todas las operaciones sintácticas (`succ`, `pred`, `add`, `mul`, `exp`) y predicados (`provable_op`, `true_op`, `list_nth_eq`) respetan la relación $\sim_\Delta$ (`succ_congruence`, `add_congruence`, etc.). Se levantan al cociente utilizando `Quotient.lift` anidado, construyendo la estructura `Model s`.

3.8. Evaluación, lema de verdad y teorema de completitud (capa 7)

* **Satisfacción tarskiana** (`satisfies`): Se define inductivamente la verdad de una fórmula en el modelo canónico bajo un entorno de asignación $\sigma : \mathbb{N} \to \text{TermClosed } s$, utilizando la función de sustitución simultánea `substAll`.

* **El lema de verdad de Tarski** (`truth_lemma`): Se demuestra por inducción estructural sobre la fórmula φ que la verdad semántica en el modelo canónico equivale de manera exacta a la pertenencia sintáctica a la teoría Δ:

$$ M(\Delta) \models_\sigma \varphi \iff \text{substAll } \varphi\ \sigma \in \Delta $$

El caso del cuantificador universal/existencial hace un uso crítico de la **propiedad de Henkin** (`LindenbaumLimit_Henkin`) y del lema de equivalencia de doble negación (`neg_forall_iff_neg_forall_double_neg`).

* **Teorema de completitud de Gödel-Henkin** (`completitud`): Se culmina probando las dos direcciones para una sentencia φ:

* *Corrección ($\Rightarrow$)*: Si $\emptyset \vdash \varphi$, por debilitamiento $\varphi \in \Delta$, y por el lema de verdad, $M(\Delta) \models \varphi$.

* *Completitud ($\Leftarrow$)*: Si $\nvdash \varphi$, entonces $\{\neg \varphi\}$ es consistente (`no_deduccion_implica_consistencia_neg`). Se construye su extensión de Henkin-Lindenbaum Δ, la cual contiene a $\neg \varphi$. Por el lema de verdad, $M(\Delta) \models \neg \varphi$, por lo que $M(\Delta) \not\models \varphi$, contradiciendo que φ fuera universalmente válida.

---

4. Innovaciones técnicas destacadas en la formalización Lean 4

El lector que examine el código de esta primera parte observará varias soluciones elegantes a problemas clásicos de formalización metamatemática:

1. **Tipos fantasma y tipado de estratos**: Las etiquetas de estrato s en `ArithTerm s` y `ArithForm s` actúan como restricciones de tipo en Lean que impiden la mezcla accidental de fórmulas de distintos niveles, mientras que funciones como `liftTerm` y `liftForm` permiten elevar de forma transparente conceptos entre estratos adyacentes.

2. **Eliminación del combustible en los decodificadores**: En lugar de requerir argumentos inductivos auxiliares de combustible (*fuel*) para la ejecución del parser, los teoremas `serializeTerm_length_ge_size` y `serializeForm_length_ge_size` convierten la propia longitud del código en un combustible físico garantizado, desacoplando la sintaxis de dependencias operacionales.

3. **Tratamiento de la alfa-conversión**: Mediante los lemas `renameBoundVar` y `rename_equiv`, se proporciona un mecanismo mecánico y constructivo para rectificar fórmulas (deshacer ligaduras conflictivas) antes de instanciar testigos de Henkin.

4. **Cero «sorry» en el núcleo inductivo**: Cada paso inductivo de las cadenas de Henkin-Lindenbaum y cada caso del lema de verdad han sido completamente cerrados dentro del kernel de Lean 4, garantizando una confianza lógica absoluta en la formalización.

---

5. Guía de lectura para la primera parte

Al explorar el archivo fuente, se recomienda seguir el siguiente orden secuencial para apreciar la construcción acumulativa de la prueba:

1. **Lemas de orden aritmético** (`lt_of_max_lt_left` a `max_le_max_of_le`): Familiarícese con el álgebra de cotas superiores para la gestión de variables.

2. **Definiciones sintácticas y encodings** (`HenkinConst` a `Code s`): Observe la dualidad entre representaciones planas (`List Nat`) y abstractas (`Code s`).

3. **Parser e inversión** (`parseTerm_serializeTerm`, `parseForm_serializeForm`, `decodeTerm_encodeTerm`, `decodeForm_encodeForm`): Verifique la biyectividad de la serialización.

4. **Sustitución y seguridad** (`substForm`, `IsFreeFor`, `NotFreeForm`, `NoBoundVar`): Tómese un tiempo en comprender el predicado de seguridad contra la captura de variables.

5. **Sistema deductivo** (`SPA_Deduction_Internal`): Examine la lista de reglas de inferencia y asegúrese de identificar cómo los operadores estratificados interactúan con la igualdad.

6. **Cotas de derivación y conservatividad de Henkin** (`FreshForDerivation`, `MaxVarDeriv`, `substHenkin_preservation_of_fresh`): Analice cómo se eliminan las constantes de Henkin para probar la conservatividad de los testigos.

7. **Secuencias de Henkin y Lindenbaum** (`GammaSeq`, `GammaLimit`, `LindenbaumSeq`, `LindenbaumLimit`): Siga la prueba inductiva de consistencia y la prueba de la propiedad de Henkin (`LindenbaumLimit_Henkin`).

8. **Modelo canónico y lema de verdad** (`termSetoid`, `M`, `succ_M`, `add_M`, `satisfies`, `truth_lemma`): Vea cómo se construye el cociente de términos cerrados y cómo la satisfacción semántica se reduce a pertenencia sintáctica.

9. **Teorema de completitud** (`completitud`): El broche de oro que unifica la sintaxis, la compacidad sintáctica, la extensión maximal y el modelo canónico.

Importaciones y soporte matemático inicial

El archivo comienza con la importación de bibliotecas fundamentales de Mathlib4:

- `Mathlib.Data.Set.Basic`: Proporciona la teoría básica de conjuntos, necesaria para trabajar con conjuntos de fórmulas como teorías formales.

- `Mathlib.Tactic`: Incluye tácticas adicionales para simplificar demostraciones.

- `Mathlib.Logic.Encodable.Basic`: Define la clase `Encodable` que establece la numerabilidad de tipos, indispensable para las enumeraciones de Henkin-Lindenbaum.

- `Mathlib.Data.Nat.ChineseRemainder`: Proporciona el Teorema Chino del Resto para números naturales, utilizado en la construcción de la función beta de Gödel.

Lemas de orden y cotas para el operador máximo

Se establecen tres lemas elementales sobre el orden de números naturales y el operador máximo, que serán utilizados como herramientas auxiliares en el manejo de variables frescas durante la construcción de Henkin.

Lema `lt_of_max_lt_left`

Establece que si el máximo de dos números naturales a y b es estrictamente menor que un tercer número c, entonces el primer componente a es también menor que c:

$$ \forall a, b, c \in \mathbb{N},\ \max(a, b) < c \implies a < c $$

Intuición y necesidad

Este lema de soporte aritmético es elemental pero esencial para la manipulación de variables y constantes. Durante la construcción de la secuencia de Henkin, se añaden constantemente fórmulas con variables y constantes cuyos índices deben mantenerse acotados para asegurar que existan suficientes símbolos "frescos" (no utilizados). Cuando una cota se define mediante el operador máximo, este lema permite proyectar dicha cota sobre los componentes individuales.

Idea de la demostración

La prueba se basa en la transitividad del orden en los números naturales: dado que $a \le \max(a, b)$ es una propiedad universal del supremo, combinando esto con la hipótesis $\max(a, b) < c$, se obtiene directamente $a < c$.

Lema `lt_of_max_lt_right`

Establece la preservación de la cota superior para el segundo componente del operador máximo:

$$ \forall a, b, c \in \mathbb{N},\ \max(a, b) < c \implies b < c $$

Intuición y necesidad

Al igual que el lema anterior, se utiliza en los pasos inductivos donde las fórmulas o términos se bifurcan en subcomponentes (por ejemplo, en la suma, multiplicación o implicación lógica). Al analizar la complejidad o las variables de estos términos compuestos, el análisis requiere descomponer la cota del máximo en cotas individuales para cada subtérmino.

Lema `max_le_max_of_le`

Establece una propiedad algebraica sobre el operador máximo de los números naturales:

$$ \forall a, b, c, d, e \in \mathbb{N},\ (a \le \max(c, e)) \land (b \le \max(d, e)) \implies \max(a, b) \le \max(\max(c, d), e) $$

Intuición y necesidad

En la manipulación de términos y fórmulas bajo sustitución, las cotas de las variables suelen estar expresadas en relación con conjuntos de variables previas y variables nuevas (representadas por e). Para unificar estas cotas en una única cota global válida para el término compuesto, es indispensable reorganizar los operadores de máximo sin perder la desigualdad. Este lema realiza esa manipulación estructural en el retículo distributivo de $(\mathbb{N}, \le, \max)$.

```
import Mathlib.Data.Set.Basic

import Mathlib.Tactic

import Mathlib.Logic.Encodable.Basic

import Mathlib.Data.Nat.ChineseRemainder



--
============================================================================

-- SOPORTE MATEMÁTICO: LEMAS DE ORDEN Y COTAS PARA MAX

--
============================================================================


lemma lt_of_max_lt_left {a b c : Nat} (h : Nat.max a b < c) : a < c :=
```

```
  Nat.lt_of_le_of_lt (Nat.le_max_left a b) h

lemma lt_of_max_lt_right {a b c : Nat} (h : Nat.max a b < c) : b < c :=
  Nat.lt_of_le_of_lt (Nat.le_max_right a b) h

lemma max_le_max_of_le {a b c d e : Nat} (h1 : a ≤ Nat.max c e) (h2 : b ≤ Nat.max d e) :
    Nat.max a b ≤ Nat.max (Nat.max c d) e := by
  apply Nat.max_le.mpr
  constructor
  · apply Nat.le_trans h1
    apply Nat.max_le.mpr
    constructor
    · have : c ≤ Nat.max c d := Nat.le_max_left c d
      exact Nat.le_trans this (Nat.le_max_left (Nat.max c d) e)
    · exact Nat.le_max_right (Nat.max c d) e
  · apply Nat.le_trans h2
    apply Nat.max_le.mpr
    constructor
    · have : d ≤ Nat.max c d := Nat.le_max_right c d
      exact Nat.le_trans this (Nat.le_max_left (Nat.max c d) e)
    · exact Nat.le_max_right (Nat.max c d) e
```

Estructuras sintácticas y codificación

Se definen las estructuras fundamentales que constituyen la sintaxis del lenguaje formal estratificado SPA. Estas definiciones establecen los tipos de datos inductivos para constantes de Henkin, numerales estratificados y términos aritméticos, junto con su infraestructura de numerabilidad.

Estructura `HenkinConst`: Constantes de Henkin

Define los símbolos de constantes de Henkin como una estructura que envuelve un número natural:

$$ \text{HenkinConst} := \{ \text{index} : \mathbb{N} \} $$

Intuición y necesidad

En el método de Henkin, el lenguaje de la teoría se expande agregando constantes llamadas "testigos" para las fórmulas existenciales. `HenkinConst` es simplemente una estructura con un único campo `index : Nat`, lo que establece una biyección constructiva con $\mathbb{N}$. Esto nos dota de un suministro infinito de constantes frescas ordenadas por su índice.

Instancia `HenkinConst_encodable`: Numerabilidad de las constantes de Henkin

Registra el hecho de que el tipo `HenkinConst` es numerable (satisface la clase de tipos `Encodable` de Mathlib):

$$ \text{Encodable}(\text{HenkinConst}) $$

Intuición y necesidad

El método de Lindenbaum y la construcción de la secuencia de Henkin requieren ordenar y procesar de manera exhaustiva todos los elementos del lenguaje (términos, constantes y fórmulas). La numerabilidad provista por la biyección de `HenkinConst` con `Nat` es el bloque de construcción básico para este fin.

Tipo inductivo `NatS`: Numerales sintácticos estratificados

Representa el conjunto de números naturales de forma sintáctica como numerales dentro del lenguaje, parametrizados por un estrato o nivel $s \in \mathbb{N}$:

$$ \text{NatS } s ::= \text{zero} \mid \text{succ}(n : \text{NatS } s) $$

Intuición y necesidad

En sistemas de lógica matemática con jerarquías de verdad (para evitar paradojas autorreferenciales), es útil etiquetar los numerales sintácticos según el estrato s en el que operan. El parámetro s en `NatS s` actúa como una **etiqueta de tipo fantasma** y no como un índice que afecte o altere la construcción de los numerales. La estructura numérica interna e inductiva (generada por `zero` y `succ`) es idéntica para todo s, pero el tipado de Lean distingue estricta y formalmente a `NatS s1` de `NatS s2` para mantener la disciplina de estratificación.

Funciones `NatS_encode`, `NatS_decode` y Lema `NatS_decode_encode`

Definen la codificación y decodificación entre los numerales sintácticos `NatS s` y los números naturales nativos de Lean, demostrando que son inversas por la izquierda:

$$ \forall n \in \text{NatS } s,\ \text{decode}(\text{encode}(n)) = n $$

Intuición y necesidad

Para demostrar la numerabilidad de `NatS s`, es necesario construir un isomorfismo efectivo con $\mathbb{N}$. El lema `NatS_decode_encode` valida inductivamente que la codificación y decodificación preservan estructuralmente el numeral original, asegurando que no se produce pérdida de información sintáctica.

Instancia `NatS_encodable`: Numerabilidad de los numerales estratificados

Declara formalmente que el tipo sintáctico `NatS s` cumple con los requisitos de numerabilidad para cualquier estrato s:

$$ \forall s \in \mathbb{N},\ \text{Encodable}(\text{NatS } s) $$

Tipo inductivo `ArithTerm`: Términos aritméticos

Define la sintaxis de los términos de la aritmética estratificada a nivel s. Un término puede ser una variable (`var`), un numeral con etiqueta de estrato (`const`), una constante de Henkin (`henkin`), el cero aritmético, o bien construirse recursivamente mediante sucesor (`succ`), suma (`add`), producto (`mul`), exponenciación (`exp`) y predecesor (`pred`):

$$ \text{ArithTerm } s ::= \text{var}(n : \mathbb{N}) \mid \text{const}(c : \text{NatS } s) \mid \text{henkin}(h : \text{HenkinConst}) \mid \text{zero} \mid \text{succ}(t) \mid \text{add}(t_1, t_2) \mid \text{mul}(t_1, t_2) \mid \text{exp}(t_1, t_2) \mid \text{pred}(t) $$

Intuición y necesidad

Constituye el núcleo del lenguaje aritmético de la teoría. Se define inductivamente como un término sintáctico libre. La inclusión de `henkin` dentro de los términos es

fundamental, ya que permite que los testigos de Henkin se comporten sintácticamente de la misma manera que las variables o numerales tradicionales dentro de las ecuaciones.

Función `termSize`: Tamaño de un término

Calcula el número total de nodos que conforman el árbol sintáctico de un término aritmético:

$$ \text{termSize} : \text{ArithTerm } s \to \mathbb{N} $$

Intuición y necesidad

A diferencia de otros desarrollos recursivos que se apoyan en inducción estructural generalizada, la función primordial de `termSize` en este archivo es puramente operacional y concreta: **medir el número de nodos del árbol sintáctico para suministrar el "combustible" (*fuel*) seguro y exacto** que requiere el decodificador recursivo en la prueba de corrección del parser. Cada constructor base (variables, constantes, Henkin, zero) tiene tamaño 1, y los constructores compuestos suman los tamaños de sus subcomponentes más 1 por el nodo raíz.

--
==========

-- ESTRUCTURAS SINTÁCTICAS Y CODIFICACIÓN

--
==========

structure HenkinConst where

  index : Nat

deriving DecidableEq

```
instance HenkinConst_encodable : Encodable HenkinConst :=
  Encodable.ofLeftInverse (fun c => c.index) (fun n => ⟨n⟩) (fun ⟨_⟩ => rfl)

inductive NatS (s : Nat) : Type where
  | zero : NatS s
  | succ : NatS s → NatS s
  deriving DecidableEq

def NatS_encode {s : Nat} : NatS s → Nat
  | NatS.zero => 0
  | NatS.succ n => NatS_encode n + 1

def NatS_decode {s : Nat} : Nat → NatS s
  | 0 => NatS.zero
  | n + 1 => NatS.succ (NatS_decode n)

lemma NatS_decode_encode {s : Nat} (n : NatS s) : NatS_decode (NatS_encode n) = n :=
by
  induction n with
  | zero => rfl
  | succ n' ih =>
    dsimp [NatS_encode, NatS_decode]
    rw [ih]

instance NatS_encodable (s : Nat) : Encodable (NatS s) :=
  Encodable.ofLeftInverse NatS_encode NatS_decode NatS_decode_encode

inductive ArithTerm (s : Nat) : Type where
```

```
  | var    : Nat → ArithTerm s
  | const  : NatS s → ArithTerm s
  | henkin : HenkinConst → ArithTerm s
  | zero   : ArithTerm s
  | succ   : ArithTerm s → ArithTerm s
  | add    : ArithTerm s → ArithTerm s → ArithTerm s
  | mul    : ArithTerm s → ArithTerm s → ArithTerm s
  | exp    : ArithTerm s → ArithTerm s → ArithTerm s
  | pred   : ArithTerm s → ArithTerm s

def termSize {s : Nat} : ArithTerm s → Nat
  | ArithTerm.var _ => 1
  | ArithTerm.const _ => 1
  | ArithTerm.henkin _ => 1
  | ArithTerm.zero => 1
  | ArithTerm.succ t => termSize t + 1
  | ArithTerm.add t1 t2 => termSize t1 + termSize t2 + 1
  | ArithTerm.mul t1 t2 => termSize t1 + termSize t2 + 1
  | ArithTerm.exp t1 t2 => termSize t1 + termSize t2 + 1
  | ArithTerm.pred t => termSize t + 1
```

Tipo inductivo `ArithForm`: Fórmulas lógicas estratificadas

Define la sintaxis de las fórmulas lógicas a nivel s. Incluye todos los constructores necesarios para expresar la aritmética y los operadores estratificados de verdad y demostrabilidad:

$$ \text{ArithForm } s ::= \ \bot \mid t_1 = t_2 \mid \varphi \to \psi \mid \neg \varphi \mid \forall v_n\ \varphi \mid \text{provable_op}_{s_{\text{lower}}}(h, t) \mid \text{true_op}_{s_{\text{lower}}}(h, t) \mid \text{list_nth_eq}(t_1, t_2, t_3) $$

Intuición y necesidad

Formaliza el lenguaje de las proposiciones lógicas de la aritmética. Para evitar las inconsistencias semánticas de la autorreferencia no restringida, los operadores `provable_op` y `true_op` se encuentran estrictamente ligados a una condición de nivel, exigiendo que la noción de verdad o prueba de un estrato se evalúe utilizando la sintaxis del nivel inmediatamente anterior. El testigo $h : s = s_{\text{lower}} + 1$ certifica esta relación jerárquica. El constructor `list_nth_eq` permite expresar relaciones sobre listas codificadas, una capacidad necesaria para ciertas codificaciones sintácticas avanzadas.

Funciones auxiliares de estratificación

Función `get_s_lower`: Descomposición de estratos

Una función auxiliar decidible que intenta descomponer un estrato s en su predecesor directo s_{lower} tal que $s = s_{\text{lower}} + 1$:

$$ \text{get_s_lower}(s) = \begin{cases} \text{none} & \text{si } s = 0 \\ \text{some } \langle s-1, \text{rfl} \rangle & \text{si } s > 0 \end{cases} $$

Intuición y necesidad

Esta función es utilizada por el parser de fórmulas para verificar dinámicamente la condición de estratificación cuando encuentra operadores `provable_op` o `true_op`. Si el estrato es 0, no puede haber un nivel inferior, por lo que estos operadores no son válidos en el nivel base.

Función `formSize`: Tamaño de una fórmula

Calcula el número total de nodos que conforman el árbol sintáctico de una fórmula lógica, análogamente a `termSize` para términos:

$$ \text{formSize} : \text{ArithForm } s \to \mathbb{N} $$

Intuición y necesidad

Al igual que `termSize`, esta función proporciona el combustible necesario para el parser de fórmulas. Los casos base (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`) tienen tamaño 1, mientras que `impl` suma los tamaños de sus subfórmulas más 1, y los constructores unarios (`neg`, `forall_form`) suman 1 al tamaño de su subfórmula.

Infraestructura de códigos restringidos (`Code s`)

Tipo inductivo `Code`: Códigos de Gödel estratificados

Se define el tipo `Code s` como una estructura de datos indexada por el estrato s, que encapsula listas de números naturales de manera abstracta:

$$ \text{Code } s ::= \text{nil} \mid \text{cons}(n : \mathbb{N},\ xs : \text{Code } s) $$

Intuición y necesidad

En la Fase 1a de la Gödelización, es necesario representar los códigos numéricos de las fórmulas y términos de manera que se respete la estratificación del lenguaje. `Code s` actúa como un tipo abstracto de datos que envuelve listas planas de naturales, pero manteniendo una distinción de tipo a nivel de Lean según el estrato s. Esta distinción fantasma impide la mezcla inadvertida de códigos pertenecientes a estratos diferentes durante la manipulación sintáctica.

Funciones `toList`, `ofList` y lema `ofList_toList`

Definen la proyección al metalenguaje y la inyección desde el metalenguaje, estableciendo un isomorfismo estructural con las listas nativas:

$$ \text{toList} : \text{Code } s \to \text{List } \mathbb{N} $$

$$ \text{ofList} : \text{List } \mathbb{N} \to \text{Code } s $$

Intuición y necesidad

Estas funciones permiten traducir entre la representación interna estratificada de los códigos y la representación plana del metalenguaje, necesaria para operaciones de serialización y codificación numérica. El lema `ofList_toList` demuestra que son inversas por la izquierda, garantizando que no hay pérdida de información al transitar entre ambas representaciones.

Instancia `Encodable (Code s)`

Registra formalmente la numerabilidad de `Code s` mediante la inyección inversible por la izquierda provista por `toList` y `ofList`:

$$ \forall s \in \mathbb{N},\ \text{Encodable}(\text{Code } s) $$

Intuición y necesidad

La numerabilidad de `Code s` es un requisito técnico indispensable para la construcción de Henkin-Lindenbaum. Al declarar `Code s` como `Encodable`, se habilita su uso en enumeraciones dentro del metalenguaje de Lean sin exponer las funciones de aplanamiento en el lenguaje objeto, preservando así la abstracción estratificada.

```
inductive ArithForm (s : Nat) : Type where
  | bottom     : ArithForm s
  | eq         : ArithTerm s → ArithTerm s → ArithForm s
  | impl       : ArithForm s → ArithForm s → ArithForm s
  | neg        : ArithForm s → ArithForm s
  | forall_form : Nat → ArithForm s → ArithForm s
  | provable_op : {s_lower : Nat} → (s = s_lower + 1) → ArithTerm s → ArithForm s
  | true_op     : {s_lower : Nat} → (s = s_lower + 1) → ArithTerm s → ArithForm s
  | list_nth_eq : ArithTerm s → ArithTerm s → ArithTerm s → ArithForm s
```

```
--
================================================================
==

--
================================================================
==
-- BLOQUE UNIFICADO Y SEGUIDO DE LAS FASES 1a Y 1b (SIN DUPLICADOS)
--
================================================================
==

--
================================================================
==


--
================================================================
==
-- 1. FUNCIONES AUXILIARES DE ESTRATIFICACIÓN
--
================================================================
==


def get_s_lower (s : Nat) : Option { s_lower : Nat // s = s_lower + 1 } :=
 match s with
 | 0 => none
 | s_lower + 1 => some ⟨s_lower, rfl⟩

def formSize {s : Nat} : ArithForm s → Nat
 | ArithForm.bottom => 1
 | ArithForm.eq _ _ => 1
```

```
  | ArithForm.impl f1 f2 => formSize f1 + formSize f2 + 1
  | ArithForm.neg f => formSize f + 1
  | ArithForm.forall_form _ f => formSize f + 1
  | ArithForm.provable_op _ _ => 1
  | ArithForm.true_op _ _ => 1
  | ArithForm.list_nth_eq _ _ _ => 1

--
================================================================
==
-- 2. INFRAESTRUCTURA DE CÓDIGOS RESTRINGIDOS (Code s) - Fase 1a
--
================================================================
==

/--
   El tipo de los códigos restringidos para el estrato s.
   Actúa como un tipo abstracto de datos indexado para la Gödelización.
-/
inductive Code (s : Nat) : Type where
  | nil  : Code s
  | cons : Nat → Code s → Code s
  deriving DecidableEq

namespace Code

/-- Función de proyección al metalenguaje (Lista plana de naturales) -/
def toList {s : Nat} : Code s → List Nat
  | Code.nil => []
  | Code.cons x xs => x :: toList xs
```

```
/-- Función de inyección desde el metalenguaje -/
def ofList {s : Nat} : List Nat → Code s
 | [] => Code.nil
 | x :: xs => Code.cons x (ofList xs)

/-- Demostración de inversión de proyección/inyección -/
theorem ofList_toList {s : Nat} (c : Code s) : ofList (toList c) = c := by
 induction c with
 | nil => rfl
 | cons x xs ih =>
  dsimp [toList, ofList]
  rw [ih]

end Code

/-- Instancia de numerabilidad Encodable para Code s -/
instance (s : Nat) : Encodable (Code s) :=
 Encodable.ofLeftInverse Code.toList Code.ofList Code.ofList_toList
```

Operaciones algebraicas y medidas sobre `Code s`

Se definen las operaciones fundamentales sobre el tipo abstracto de códigos `Code s`: concatenación, medición de longitud, y sus propiedades algebraicas.

Función `Code.append`: Concatenación de códigos

Define la operación de concatenación para `Code s`, análoga a la concatenación de listas pero respetando la indexación fantasma del estrato:

$$ \text{append} : \text{Code } s \to \text{Code } s \to \text{Code } s $$

$$ \text{append}(\text{nil}, ys) = ys $$

$$ \text{append}(\text{cons}(x, xs), ys) = \text{cons}(x, \text{append}(xs, ys)) $$

Función `Code.length`: Longitud de un código

Define la longitud de un código como el número de nodos `cons` en su estructura inductiva:

$$ \text{length} : \text{Code } s \to \mathbb{N} $$

$$ \text{length}(\text{nil}) = 0 $$

$$ \text{length}(\text{cons}(x, xs)) = \text{length}(xs) + 1 $$

Lema `Code.append_assoc`: Asociatividad de la concatenación

$$ \forall xs, ys, zs \in \text{Code } s,\ \text{append}(\text{append}(xs, ys), zs) = \text{append}(xs, \text{append}(ys, zs)) $$

La demostración procede por inducción sobre xs, utilizando la definición recursiva de `append`.

Lema `Code.length_append`: Homomorfismo de longitud

$$ \forall xs, ys \in \text{Code } s,\ \text{length}(\text{append}(xs, ys)) = \text{length}(xs) + \text{length}(ys) $$

La longitud es aditiva respecto a la concatenación. La demostración es por inducción sobre xs con aritmética auxiliar.

Lema `Code.append_nil`: Elemento neutro por la derecha

$$ \forall xs \in \text{Code } s,\ \text{append}(xs, \text{nil}) = xs $$

Adaptación de la sintaxis de términos sobre `Code s`

Se redefine la serialización y el parser de términos para que operen sobre el tipo abstracto `Code s` en lugar de listas planas de números naturales. Esta es la culminación de la Fase 1b.

Función `serializeTerm`: Serialización de términos sobre `Code s`

Convierte un término aritmético en su representación como código estratificado, asignando etiquetas numéricas a cada constructor:

$$ \text{serializeTerm} : \text{ArithTerm } s \to \text{Code } s $$

Constructor	Etiqueta	Estructura del código
`var n`	0	`[0, n]`
`const c`	1	`[1, encode(c)]`
`henkin h`	2	`[2, h.index]`
`zero`	3	`[3]`
`succ t`	4	`4 :: serializeTerm(t)`
`add t1 t2`	5	`5 :: (serializeTerm(t1) ++ serializeTerm(t2))`
`mul t1 t2`	6	`6 :: (serializeTerm(t1) ++ serializeTerm(t2))`
`exp t1 t2`	7	`7 :: (serializeTerm(t1) ++ serializeTerm(t2))`
`pred t`	8	`8 :: serializeTerm(t)`

Función `parseTerm`: Parser de términos con combustible sobre `Code s`

Decodifica un código estratificado recuperando el término original, utilizando un parámetro de combustible (`fuel`) que decrece en cada paso recursivo para garantizar la terminación:

$$ \text{parseTerm} : \mathbb{N} \to \text{Code } s \to \text{Option} (\text{ArithTerm } s \times \text{Code } s) $$

La estructura es paralela a la del parser sobre listas, pero adaptada al tipo `Code s`. Cada caso lee la etiqueta del constructor y procede recursivamente según corresponda. Los constructores binarios (`add`, `mul`, `exp`) requieren dos llamadas recursivas secuenciales, mientras que los unarios (`succ`, `pred`) requieren una sola. El combustible se consume en cada llamada recursiva, y si se agota, el parser retorna `none`.

```
--
================================================================
==
-- 3. OPERACIONES ALGEBRAICAS Y MEDIDAS SOBRE CODE S
--
================================================================
==

/-- Concatenación de códigos en el estrato s -/
def Code.append {s : Nat} : Code s → Code s → Code s
  | Code.nil, ys => ys
  | Code.cons x xs, ys => Code.cons x (Code.append xs ys)

/-- Medición de longitud (fuel) sobre el tipo de códigos Code s -/
def Code.length {s : Nat} : Code s → Nat
  | Code.nil => 0
  | Code.cons _ xs => Code.length xs + 1

/-- Ley de asociatividad para la concatenación de códigos -/
lemma Code.append_assoc {s : Nat} (xs ys zs : Code s) :
    Code.append (Code.append xs ys) zs = Code.append xs (Code.append ys zs) := by
  induction xs with
```

```
  | nil => rfl
  | cons x xs ih =>
    simp only [Code.append]
    rw [ih]

/-- Homomorfismo de longitud respecto a la concatenación Code.append -/
lemma Code.length_append {s : Nat} (xs ys : Code s) :
    (Code.append xs ys).length = xs.length + ys.length := by
  induction xs with
  | nil =>
    simp only [Code.append, Code.length]
    rw [Nat.zero_add]
  | cons x xs ih =>
    simp only [Code.append, Code.length]
    rw [ih]
    omega

/-- Ley de identidad por la derecha para la concatenación -/
lemma Code.append_nil {s : Nat} (xs : Code s) :
    Code.append xs Code.nil = xs := by
  induction xs with
  | nil => rfl
  | cons x xs ih =>
    simp only [Code.append]
    rw [ih]

--
==========================================================================
==
```

```
-- 4. ADAPTACIÓN DE LA SINTAXIS DE TÉRMINOS (ArithTerm s) - Fase 1b

--
==========================================================================
==


/-- Redefinición de la serialización de términos sobre Code s -/

def serializeTerm {s : Nat} : ArithTerm s → Code s

  | ArithTerm.var n => Code.cons 0 (Code.cons n Code.nil)

  | ArithTerm.const c => Code.cons 1 (Code.cons (NatS_encode c) Code.nil)

  | ArithTerm.henkin c => Code.cons 2 (Code.cons c.index Code.nil)

  | ArithTerm.zero => Code.cons 3 Code.nil

  | ArithTerm.succ t => Code.cons 4 (serializeTerm t)

  | ArithTerm.add t1 t2 => Code.cons 5 (Code.append (serializeTerm t1)
(serializeTerm t2))

  | ArithTerm.mul t1 t2 => Code.cons 6 (Code.append (serializeTerm t1)
(serializeTerm t2))

  | ArithTerm.exp t1 t2 => Code.cons 7 (Code.append (serializeTerm t1) (serializeTerm
t2))

  | ArithTerm.pred t => Code.cons 8 (serializeTerm t)


/-- Redefinición del parser de términos con combustible sobre Code s -/

def parseTerm {s : Nat} : Nat → Code s → Option (ArithTerm s × Code s)

  | 0, _ => none

  | _ + 1, Code.nil => none

  | _ + 1, Code.cons 0 (Code.cons n L') => some (ArithTerm.var n, L')

  | _ + 1, Code.cons 1 (Code.cons c_code L') => some (ArithTerm.const (NatS_decode
c_code), L')

  | _ + 1, Code.cons 2 (Code.cons h_code L') => some (ArithTerm.henkin ⟨h_code⟩, L')

  | _ + 1, Code.cons 3 L' => some (ArithTerm.zero, L')

  | f + 1, Code.cons 4 L' =>
```

```
  match parseTerm f L' with
  | some (t, L'') => some (ArithTerm.succ t, L'')
  | none => none
| f + 1, Code.cons 5 L' =>
  match parseTerm f L' with
  | some (t1, L1) =>
    match parseTerm f L1 with
    | some (t2, L2) => some (ArithTerm.add t1 t2, L2)
    | none => none
  | none => none
| f + 1, Code.cons 6 L' =>
  match parseTerm f L' with
  | some (t1, L1) =>
    match parseTerm f L1 with
    | some (t2, L2) => some (ArithTerm.mul t1 t2, L2)
    | none => none
  | none => none
| f + 1, Code.cons 7 L' =>
  match parseTerm f L' with
  | some (t1, L1) =>
    match parseTerm f L1 with
    | some (t2, L2) => some (ArithTerm.exp t1 t2, L2)
    | none => none
  | none => none
| f + 1, Code.cons 8 L' =>
  match parseTerm f L' with
  | some (t, L'') => some (ArithTerm.pred t, L'')
  | none => none
```

```
  | _ + 1, _ => none
```

Lema `parseTerm_serializeTerm`: Corrección del parser de términos sobre `Code s`

Se demuestra el teorema fundamental de corrección del parser: para cualquier término t, si se proporciona suficiente combustible (al menos el tamaño de t), el parser aplicado a la serialización de t concatenada con cualquier lista remanente L recupera exactamente t y devuelve L como resto:

$$ \forall t \in \text{ArithTerm } s,\ \forall \text{fuel} \ge \text{termSize}(t),\ \forall L \in \text{Code } s, $$

$$ \text{parseTerm}(\text{fuel}, \text{append}(\text{serializeTerm}(t), L)) = \text{some}(t, L) $$

Estructura de la demostración

La prueba procede por inducción estructural sobre t, cubriendo todos los constructores de `ArithTerm`:

Casos base (`var`, `const`, `henkin`, `zero`): Todos tienen tamaño 1. Se verifica que el combustible debe ser al menos 1, por lo que `fuel` es de la forma `f + 1`. La simplificación de `serializeTerm`, `Code.append` y `parseTerm` produce directamente el resultado esperado. Para el caso `const`, se utiliza adicionalmente el lema `NatS_decode_encode` para cancelar la codificación y decodificación.

Casos unarios (`succ`, `pred`): El tamaño es `termSize(t') + 1`. Se requiere que el combustible sea al menos este valor. Tras consumir un paso de combustible, se aplica la hipótesis de inducción con el combustible restante (que es al menos `termSize(t')`) para procesar el subtérmino.

Casos binarios (`add`, `mul`, `exp`): El tamaño es `termSize(t1) + termSize(t2) + 1`. La clave técnica aquí es la asociatividad de `Code.append`: la serialización de un término binario produce un código de la forma `cons(etiqueta, append(serializeTerm(t1), serializeTerm(t2)))`. Al concatenar con L, se utiliza `Code.append_assoc` para reagrupar como `append(serializeTerm(t1), append(serializeTerm(t2), L))`, lo que permite aplicar primero la hipótesis de inducción para t_1 (con combustible $f \ge \text{termSize}(t_1)$) y luego la de t_2 (con el mismo combustible $f \ge \text{termSize}(t_2)$), procesando secuencialmente ambos subtérminos.

Significado

Este lema establece la corrección y completitud del parser respecto a la serialización definida. Garantiza que el proceso de codificación-decodificación es una biyección entre términos y sus representaciones serializadas, siempre que se provea suficiente combustible. La utilización de `Code.append_assoc` para reordenar la concatenación en los casos binarios es un detalle técnico crucial que permite que las hipótesis de inducción se apliquen en el orden correcto. Este resultado es la base sobre la cual se construye la instancia `Encodable` para `ArithTerm s`, ya que proporciona la inversa por la izquierda necesaria para establecer la numerabilidad.

```
/-- Lema de inversión del parser de términos sobre Code s -/
lemma parseTerm_serializeTerm {s : Nat} (t : ArithTerm s) : ∀ (fuel : Nat), fuel ≥ termSize t → ∀ (L : Code s),
  parseTerm fuel (Code.append (serializeTerm t) L) = some (t, L) := by
 induction t with
 | var n =>
  intro fuel h L
  change fuel ≥ 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
   simp only [serializeTerm, Code.append, parseTerm]
 | const c =>
  intro fuel h L
  change fuel ≥ 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
   simp only [serializeTerm, Code.append, parseTerm]
   rw [NatS_decode_encode]
 | henkin c =>
  intro fuel h L
  change fuel ≥ 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
   simp only [serializeTerm, Code.append, parseTerm]
```

```
| zero =>
  intro fuel h L
  change fuel ≥ 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
    simp only [serializeTerm, Code.append, parseTerm]
| succ t' ih =>
  intro fuel h L
  change fuel ≥ termSize t' + 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
    have h_f : f ≥ termSize t' := by omega
    simp only [serializeTerm, Code.append, parseTerm]
    rw [ih f h_f L]
| add t1 t2 ih1 ih2 =>
  intro fuel h L
  change fuel ≥ termSize t1 + termSize t2 + 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
    have h1 : f ≥ termSize t1 := by omega
    have h2 : f ≥ termSize t2 := by omega
    simp only [serializeTerm, Code.append, parseTerm]
    rw [Code.append_assoc]
    rw [ih1 f h1 (Code.append (serializeTerm t2) L)]
    dsimp only
```

```
    rw [ih2 f h2 L]
  | mul t1 t2 ih1 ih2 =>
    intro fuel h L
    change fuel ≥ termSize t1 + termSize t2 + 1 at h
    cases fuel with
    | zero => omega
    | succ f =>
      have h1 : f ≥ termSize t1 := by omega
      have h2 : f ≥ termSize t2 := by omega
      simp only [serializeTerm, Code.append, parseTerm]
      rw [Code.append_assoc]
      rw [ih1 f h1 (Code.append (serializeTerm t2) L)]
      dsimp only
      rw [ih2 f h2 L]
  | exp t1 t2 ih1 ih2 =>
    intro fuel h L
    change fuel ≥ termSize t1 + termSize t2 + 1 at h
    cases fuel with
    | zero => omega
    | succ f =>
      have h1 : f ≥ termSize t1 := by omega
      have h2 : f ≥ termSize t2 := by omega
      simp only [serializeTerm, Code.append, parseTerm]
      rw [Code.append_assoc]
      rw [ih1 f h1 (Code.append (serializeTerm t2) L)]
      dsimp only
      rw [ih2 f h2 L]
  | pred t' ih =>
```

```
  intro fuel h L
  change fuel ≥ termSize t' + 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
    have h_f : f ≥ termSize t' := by omega
    simp only [serializeTerm, Code.append, parseTerm]
    rw [ih f h_f L]
```

Lemas de acotación, decodificador y numerabilidad para `ArithTerm s`

Se completa la infraestructura de codificación para términos aritméticos estableciendo la relación entre la longitud del código serializado y el tamaño sintáctico, definiendo un decodificador de alto nivel y registrando formalmente la numerabilidad del tipo.

Lema `serializeTerm_length_ge_size`: Acotación de la longitud del código serializado

Establece que la longitud del código producido por `serializeTerm` es siempre mayor o igual que el tamaño sintáctico del término:

$$ \forall t \in \text{ArithTerm } s,\ \text{length}(\text{serializeTerm}(t)) \ge \text{termSize}(t) $$

Estructura de la demostración

La prueba procede por inducción estructural sobre t:

- **Casos base (`var`, `const`, `henkin`):** La serialización produce un código de longitud 2, mientras que `termSize` es 1. La desigualdad $2 \ge 1$ es inmediata.

- **Caso `zero`:** La serialización produce un código de longitud 1, igual al tamaño sintáctico.

- **Casos unarios (`succ`, `pred`):** La longitud del código es `length(serializeTerm(t')) + 1`, y por hipótesis de inducción `length(serializeTerm(t')) ≥ termSize(t')`. Sumando 1 a ambos lados se obtiene la desigualdad requerida.

- **Casos binarios (`add`, `mul`, `exp`):** La longitud del código es `length(serializeTerm(t1)) + length(serializeTerm(t2)) + 1`, gracias al lema `Code.length_append`. Por hipótesis de inducción, cada sumando es al menos el tamaño del subtérmino correspondiente, y la suma más 1 es al menos `termSize(t1) + termSize(t2) + 1`.

Intuición y necesidad

Este teorema permite desacoplar al parser del combustible artificial de tipo manual. Al demostrar que la longitud física de la lista serializada es un límite superior válido para `termSize(t)`, podemos utilizar la propia longitud de la lista como combustible en el decodificador de alto nivel, garantizando que nunca se agotará antes de completar el parseo.

Función `decodeTerm`: Decodificador de términos de alto nivel

Encapsula la decodificación de códigos en un término aritmético sin requerir combustible explícito, utilizando la longitud del código como combustible:

$$ \text{decodeTerm}(L) = \begin{cases} \text{some } t & \text{si } \text{parseTerm}(\text{length}(L), L) = \text{some}(t, \text{nil}) \\ \text{none} & \text{en otro caso} \end{cases} $$

Lema `decodeTerm_encodeTerm`: Corrección del decodificador

Demuestra que el decodificador recupera perfectamente el término serializado:

$$ \forall t \in \text{ArithTerm } s,\ \text{decodeTerm}(\text{serializeTerm}(t)) = \text{some } t $$

La demostración utiliza `serializeTerm_length_ge_size` para garantizar que `length(serializeTerm(t)) ≥ termSize(t)`, lo que permite aplicar `parseTerm_serializeTerm` con ese combustible. La lista remanente debe ser `nil` (código vacío), y `Code.append_nil` simplifica la expresión.

Instancia `ArithTerm_encodable`: Numerabilidad de los términos aritméticos

Registra formalmente que `ArithTerm s` es numerable para cualquier estrato s:

$$ \forall s \in \mathbb{N},\ \text{Encodable}(\text{ArithTerm } s) $$

La instancia se construye mediante `Encodable.ofLeftInjection` utilizando `serializeTerm` como inyección, `decodeTerm` como inversa por la izquierda, y `decodeTerm_encodeTerm` como prueba de la propiedad de inversión.

Intuición y necesidad

La numerabilidad de `ArithTerm s` es un requisito indispensable para la construcción de Henkin-Lindenbaum, ya que permite enumerar todos los términos del lenguaje para procesarlos secuencialmente. Junto con la numerabilidad de `ArithForm s` (definida más adelante), completa la infraestructura necesaria para las enumeraciones exhaustivas requeridas por el método de completitud.

/-- La longitud del código serializado acota superiormente al tamaño sintáctico -/

lemma serializeTerm_length_ge_size {s : Nat} (t : ArithTerm s) : (serializeTerm t).length ≥ termSize t := by

 induction t with

 | var n =>

  change 2 ≥ 1

  omega

 | const c =>

  change 2 ≥ 1

  omega

 | henkin c =>

  change 2 ≥ 1

  omega

 | zero =>

```
    change 1 ≥ 1
    omega
  | succ t' ih =>
    unfold serializeTerm termSize Code.length
    omega
  | add t1 t2 ih1 ih2 =>
    unfold serializeTerm termSize Code.length
    rw [Code.length_append]
    omega
  | mul t1 t2 ih1 ih2 =>
    unfold serializeTerm termSize Code.length
    rw [Code.length_append]
    omega
  | exp t1 t2 ih1 ih2 =>
    unfold serializeTerm termSize Code.length
    rw [Code.length_append]
    omega
  | pred t' ih =>
    unfold serializeTerm termSize Code.length
    omega

/-- Decodificador de términos de alto nivel libre de combustible artificial -/
def decodeTerm {s : Nat} (L : Code s) : Option (ArithTerm s) :=
  match parseTerm L.length L with
  | some (t, Code.nil) => some t
  | _ => none

/-- Teorema de inversión del decodificador de términos -/
```

```
lemma decodeTerm_encodeTerm {s : Nat} (t : ArithTerm s) : decodeTerm (serializeTerm t) = some t := by
  simp only [decodeTerm]
  have h_len := serializeTerm_length_ge_size t
  have h_parse := parseTerm_serializeTerm t (serializeTerm t).length h_len Code.nil
  rw [Code.append_nil] at h_parse
  rw [h_parse]

/-- Registro de la numerabilidad para ArithTerm s utilizando Code s como pivote modal -/
instance ArithTerm_encodable (s : Nat) : Encodable (ArithTerm s) :=
  Encodable.ofLeftInjection serializeTerm decodeTerm decodeTerm_encodeTerm
```

Adaptación de la sintaxis de fórmulas sobre `Code s`

Se completa la Fase 1b con la redefinición de la serialización y el parser de fórmulas sobre el tipo abstracto `Code s`, junto con el teorema de corrección correspondiente.

Función `serializeForm`: Serialización de fórmulas sobre `Code s`

Convierte una fórmula lógica en su representación como código estratificado, asignando etiquetas numéricas a cada constructor:

$$ \text{serializeForm} : \text{ArithForm } s \to \text{Code } s $$

| Constructor | Etiqueta | Estructura del código |

|---|---|---|

| `bottom` | 0 | `[0]` |

| `eq t1 t2` | 1 | `[1, encode(t1), encode(t2)]` |

| `impl f1 f2` | 2 | `2 :: (serializeForm(f1) ++ serializeForm(f2))` |

| `neg f` | 3 | `3 :: serializeForm(f)` |

| `forall_form n f` | 4 | `4 :: n :: serializeForm(f)` |

| `provable_op h t` | 5 | `[5, encode(t)]` |

| `true_op h t` | 6 | `[6, encode(t)]` |

| `list_nth_eq t1 t2 t3` | 7 | `[7, encode(t1), encode(t2), encode(t3)]` |

Para los constructores que contienen términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`), se utiliza `Encodable.encode` para obtener la representación numérica del término, aprovechando la instancia `ArithTerm_encodable` definida previamente. Los constructores recursivos (`impl`, `neg`, `forall_form`) concatenan los códigos de sus subfórmulas.

Función `parseForm`: Parser de fórmulas con combustible sobre `Code s`

Decodifica un código estratificado recuperando la fórmula original, con un parámetro de combustible que garantiza la terminación:

$$ \text{parseForm} : \mathbb{N} \to \text{Code } s \to \text{Option} (\text{ArithForm } s \times \text{Code } s) $$

La estructura refleja la del parser de términos, adaptada a los constructores de `ArithForm`. Los casos notables incluyen:

- **Operadores estratificados (`provable_op`, `true_op`):** Se verifica dinámicamente la condición de estratificación mediante `get_s_lower`. Si el estrato s no puede descomponerse como $s_{\text{lower}} + 1$ (es decir, $s = 0$), el parser rechaza la entrada.

- **Constructores con términos:** Se utiliza `Encodable.decode` para recuperar los términos a partir de sus códigos numéricos.

- **Caso recursivo `impl`:** Requiere dos llamadas recursivas secuenciales, similar a los constructores binarios de términos.

Lema `parseForm_serializeForm`: Corrección del parser de fórmulas

Establece el teorema fundamental de corrección: para cualquier fórmula φ, con suficiente combustible, el parser recupera exactamente φ:

$$ \forall \varphi \in \text{ArithForm } } s,\ \forall \text{fuel} \ge \text{formSize}(\varphi),\ \forall L \in \text{Code } } s, $$

$$ \text{parseForm}(\text{fuel}, \text{append}(\text{serializeForm}(\varphi), L)) = \text{some}(\varphi, L) $$

Estructura de la demostración

La prueba procede por inducción sobre φ, cubriendo los ocho constructores:

- **Casos base no recursivos (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`):** Todos tienen `formSize = 1`. Se verifica que el combustible es al menos 1, y la simplificación directa produce el resultado. Para `eq` y los casos con términos, se utiliza `Encodable.encodek` para cancelar la codificación-decodificación.

- **Casos unarios (`neg`, `forall_form`):** El tamaño es `formSize(f) + 1`. Tras consumir un paso de combustible, se aplica la hipótesis de inducción con el combustible restante.

- **Caso binario (`impl`):** Similar a los constructores binarios de términos: se utiliza `Code.append_assoc` para reagrupar la concatenación y aplicar secuencialmente las hipótesis de inducción para ambas subfórmulas.

- **Casos con testigo de estratificación (`provable_op`, `true_op`):** La hipótesis `h_eq : s = s_lower + 1` se elimina mediante `cases h_eq`, lo que sustituye s por $s_{\text{lower}} + 1$ y permite que `get_s_lower` devuelva exitosamente el predecesor.

Significado

Este lema, junto con su análogo para términos, completa la verificación de que la serialización y el parser forman una biyección entre las fórmulas y sus representaciones codificadas. Es el último paso antes de establecer la numerabilidad de `ArithForm s` y constituye la base sobre la cual se construye toda la maquinaria de Gödelización del sistema SPA. La corrección del parser garantiza que las operaciones sintácticas pueden ser fielmente representadas como manipulaciones numéricas sobre códigos, lo cual es el fundamento mismo de la aritmetización de la sintaxis.

```
--
==========================================================================
==

-- 5. ADAPTACIÓN DE LA SINTAXIS DE FÓRMULAS (ArithForm s) - Fase 1b

--
==========================================================================
==


/-- Redefinición de la serialización de fórmulas sobre Code s -/

def serializeForm {s : Nat} : ArithForm s → Code s

 | ArithForm.bottom => Code.cons 0 Code.nil

 | ArithForm.eq t1 t2 => Code.cons 1 (Code.cons (Encodable.encode t1) (Code.cons
(Encodable.encode t2) Code.nil))

 | ArithForm.impl f1 f2 => Code.cons 2 (Code.append (serializeForm f1)
(serializeForm f2))

 | ArithForm.neg f => Code.cons 3 (serializeForm f)

 | ArithForm.forall_form n f => Code.cons 4 (Code.cons n (serializeForm f))

 | ArithForm.provable_op _ t => Code.cons 5 (Code.cons (Encodable.encode t)
Code.nil) -- (línea 445 corregida)

 | ArithForm.true_op _ t => Code.cons 6 (Code.cons (Encodable.encode t) Code.nil)  -
- (línea 446 corregida)

 | ArithForm.list_nth_eq t1 t2 t3 => Code.cons 7 (Code.cons (Encodable.encode t1)
(Code.cons (Encodable.encode t2) (Code.cons (Encodable.encode t3) Code.nil)))




/-- Redefinición del parser de fórmulas con combustible sobre Code s -/

def parseForm {s : Nat} : Nat → Code s → Option (ArithForm s × Code s)

 | 0, _ => none

 | _ + 1, Code.nil => none
```

```
  | _ + 1, Code.cons 0 L' => some (ArithForm.bottom, L')
  | _ + 1, Code.cons 1 (Code.cons t1_code (Code.cons t2_code L')) =>
    match (Encodable.decode t1_code : Option (ArithTerm s)), (Encodable.decode t2_code : Option (ArithTerm s)) with
    | some t1, some t2 => some (ArithForm.eq t1 t2, L')
    | _, _ => none
  | f + 1, Code.cons 2 L' =>
    match parseForm f L' with
    | some (f1, L1) =>
      match parseForm f L1 with
      | some (f2, L2) => some (ArithForm.impl f1 f2, L2)
      | none => none
    | none => none
  | f + 1, Code.cons 3 L' =>
    match parseForm f L' with
    | some (f1, L'') => some (ArithForm.neg f1, L'')
    | none => none
  | f + 1, Code.cons 4 (Code.cons n L') =>
    match parseForm f L' with
    | some (f1, L'') => some (ArithForm.forall_form n f1, L'')
    | none => none
  | _ + 1, Code.cons 5 (Code.cons t_code L') =>
    match (get_s_lower s) with
    | some ⟨_, h_eq⟩ =>
      match (Encodable.decode t_code : Option (ArithTerm s)) with
      | some t => some (ArithForm.provable_op h_eq t, L')
      | none => none
    | none => none
```

```
 | _ + 1, Code.cons 6 (Code.cons t_code L') =>
   match (get_s_lower s) with
   | some ⟨_, h_eq⟩ =>
     match (Encodable.decode t_code : Option (ArithTerm s)) with
     | some t => some (ArithForm.true_op h_eq t, L')
     | none => none
   | none => none
 | _ + 1, Code.cons 7 (Code.cons t1_code (Code.cons t2_code (Code.cons t3_code L'))) =>
   match (Encodable.decode t1_code : Option (ArithTerm s)), (Encodable.decode t2_code : Option (ArithTerm s)), (Encodable.decode t3_code : Option (ArithTerm s)) with
   | some t1, some t2, some t3 => some (ArithForm.list_nth_eq t1 t2 t3, L')
   | _, _, _ => none
 | _ + 1, _ => none


/-- Lema de inversión del parser de fórmulas sobre Code s -/
lemma parseForm_serializeForm {s : Nat} (φ : ArithForm s) : ∀ (fuel : Nat), fuel ≥ formSize φ → ∀ (L : Code s),
   parseForm fuel (Code.append (serializeForm φ) L) = some (φ, L) := by
 induction φ with
 | bottom =>
   intro fuel h L
   change fuel ≥ 1 at h
   cases fuel with
   | zero => omega
   | succ f =>
     simp only [serializeForm, Code.append, parseForm]
 | eq t1 t2 =>
   intro fuel h L
```

```
  change fuel ≥ 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
    simp only [serializeForm, Code.append, parseForm, Encodable.encodek]
| impl f1 f2 ih1 ih2 =>
  intro fuel h L
  change fuel ≥ formSize f1 + formSize f2 + 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
    have h1 : f ≥ formSize f1 := by omega
    have h2 : f ≥ formSize f2 := by omega
    simp only [serializeForm, Code.append, parseForm]
    rw [Code.append_assoc]
    rw [ih1 f h1 (Code.append (serializeForm f2) L)]
    dsimp only
    rw [ih2 f h2 L]
| neg f' ih =>
  intro fuel h L
  change fuel ≥ formSize f' + 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
    have h_f : f ≥ formSize f' := by omega
    simp only [serializeForm, Code.append, parseForm]
    rw [ih f h_f L]
| forall_form n f' ih =>
```

```
  intro fuel h L
  change fuel ≥ formSize f' + 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
    have h_f : f ≥ formSize f' := by omega
    simp only [serializeForm, Code.append, parseForm]
    rw [ih f h_f L]
 | provable_op h_eq t =>
  intro fuel h L
  cases h_eq
  change fuel ≥ 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
   simp only [serializeForm, Code.append, parseForm, get_s_lower,
Encodable.encodek]
 | true_op h_eq t =>
  intro fuel h L
  cases h_eq
  change fuel ≥ 1 at h
  cases fuel with
  | zero => omega
  | succ f =>
   simp only [serializeForm, Code.append, parseForm, get_s_lower,
Encodable.encodek]
 | list_nth_eq t1 t2 t3 =>
  intro fuel h L
  change fuel ≥ 1 at h
```

```
  cases fuel with
  | zero => omega
  | succ f =>
    simp only [serializeForm, Code.append, parseForm, Encodable.encodek]
```

Lemas de acotación, decodificador y numerabilidad para `ArithForm s`

Se completa la infraestructura de codificación para fórmulas lógicas, estableciendo los resultados análogos a los de términos: relación entre longitud y tamaño, decodificador de alto nivel, y numerabilidad.

Lema `serializeForm_length_ge_size`: Acotación de la longitud del código serializado

Establece que la longitud del código producido por `serializeForm` es siempre mayor o igual que el tamaño sintáctico de la fórmula:

$$ \forall \varphi \in \text{ArithForm } s,\ \text{length}(\text{serializeForm}(\varphi)) \ge \text{formSize}(\varphi) $$

Estructura de la demostración

La prueba procede por inducción estructural sobre φ:

- **Casos base atómicos (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`):** La longitud del código serializado es una constante pequeña (1, 2, 3 o 4), mientras que `formSize` es 1. La desigualdad es inmediata por aritmética.

- **Caso `impl`:** La longitud es `length(serializeForm(f1)) + length(serializeForm(f2)) + 1` (por `Code.length_append`), y por hipótesis de inducción cada sumando es al menos `formSize(f1)` y `formSize(f2)` respectivamente. Sumando 1 se obtiene la cota requerida para `formSize(f1) + formSize(f2) + 1`.

- **Caso `neg`:** La longitud es `length(serializeForm(f)) + 1`, y por hipótesis de inducción esto es al menos `formSize(f) + 1`.

- **Caso `forall_form`:** La longitud es `length(serializeForm(f)) + 2` (por la etiqueta y el índice de variable), mientras que `formSize` es `formSize(f) + 1`. La cota se satisface holgadamente.

Intuición y necesidad

Al igual que su análogo para términos, este lema permite utilizar la longitud del código serializado como combustible automático para el decodificador, eliminando la necesidad de especificar manualmente un parámetro de combustible.

Función `decodeForm`: Decodificador de fórmulas de alto nivel

Define la decodificación de códigos en fórmulas sin requerir combustible explícito:

$$ \text{decodeForm}(L) = \begin{cases} \text{some } \varphi & \text{si } \text{parseForm}(\text{length}(L), L) = \text{some}(\varphi, \text{nil}) \\ \text{none} & \text{en otro caso} \end{cases} $$

Lema `decodeForm_encodeForm`: Corrección del decodificador

Demuestra que el decodificador recupera perfectamente la fórmula serializada:

$$ \forall \varphi \in \text{ArithForm } s,\ \text{decodeForm}(\text{serializeForm}(\varphi)) = \text{some } \varphi $$

La demostración es idéntica en estructura a `decodeTerm_encodeTerm`: se utiliza `serializeForm_length_ge_size` para garantizar combustible suficiente, `parseForm_serializeForm` para la corrección del parser, y `Code.append_nil` para simplificar la lista vacía.

Instancia `ArithForm_encodable`: Numerabilidad de las fórmulas

Registra formalmente que `ArithForm s` es numerable para cualquier estrato s:

$$ \forall s \in \mathbb{N},\ \text{Encodable}(\text{ArithForm } s) $$

La instancia se construye con `Encodable.ofLeftInjection` utilizando `serializeForm` como inyección, `decodeForm` como inversa por la izquierda, y `decodeForm_encodeForm` como prueba de inversión.

Significado global de la infraestructura de codificación

Con estas definiciones y teoremas, se completa el aparato de Gödelización para el sistema SPA:

1. **Tipos sintácticos:** `ArithTerm s` y `ArithForm s` definen la sintaxis completa del lenguaje.

2. **Serialización:** `serializeTerm` y `serializeForm` convierten objetos sintácticos en códigos abstractos `Code s`.

3. **Parseo:** `parseTerm` y `parseForm` recuperan los objetos sintácticos a partir de los códigos.

4. **Corrección:** Los lemas `parseTerm_serializeTerm` y `parseForm_serializeForm` garantizan que la serialización y el parseo son inversos.

5. **Acotación:** Los lemas `serializeTerm_length_ge_size` y `serializeForm_length_ge_size` permiten eliminar el combustible artificial.

6. **Decodificación:** `decodeTerm` y `decodeForm` proporcionan interfaces de alto nivel.

7. **Numerabilidad:** Las instancias `ArithTerm_encodable` y `ArithForm_encodable` habilitan las enumeraciones necesarias para Henkin-Lindenbaum.

Esta arquitectura en capas —tipos abstractos `Code s`, serialización, parseo con combustible, decodificación sin combustible, y numerabilidad— constituye el fundamento técnico sobre el cual se construye toda la aritmetización de la sintaxis en el sistema SPA, desde la representabilidad de funciones básicas hasta los teoremas de incompletitud.

```
/-- La longitud del código serializado acota superiormente al tamaño de la fórmula -/

lemma serializeForm_length_ge_size {s : Nat} (φ : ArithForm s) : (serializeForm φ).length ≥ formSize φ := by

  induction φ with

  | bottom =>

    change 1 ≥ 1

    omega

  | eq t1 t2 =>

    change 3 ≥ 1

    omega

  | impl f1 f2 ih1 ih2 =>

    change (Code.append (serializeForm f1) (serializeForm f2)).length + 1 ≥ formSize f1 + formSize f2 + 1

    rw [Code.length_append]

    omega

  | neg f' ih =>

    change (serializeForm f').length + 1 ≥ formSize f' + 1

    omega

  | forall_form n f' ih =>

    change (serializeForm f').length + 2 ≥ formSize f' + 1

    omega

  | provable_op h_eq t =>

    change 2 ≥ 1

    omega

  | true_op h_eq t =>
```

```
  change 2 ≥ 1
  omega
| list_nth_eq t1 t2 t3 =>
  change 4 ≥ 1
  omega

/-- Decodificador de fórmulas de alto nivel libre de combustible artificial -/
def decodeForm {s : Nat} (L : Code s) : Option (ArithForm s) :=
 match parseForm L.length L with
 | some (φ, Code.nil) => some φ
 | _ => none

/-- Teorema de inversión del decodificador de fórmulas -/
lemma decodeForm_encodeForm {s : Nat} (φ : ArithForm s) : decodeForm (serializeForm φ) = some φ := by
 simp only [decodeForm]
 have h_len := serializeForm_length_ge_size φ
 have h_parse := parseForm_serializeForm φ (serializeForm φ).length h_len Code.nil
 rw [Code.append_nil] at h_parse
 rw [h_parse]

/-- Registro de la numerabilidad para ArithForm s utilizando Code s como pivote modal -/
instance ArithForm_encodable (s : Nat) : Encodable (ArithForm s) :=
 Encodable.ofLeftInjection serializeForm decodeForm decodeForm_encodeForm
```

Predicados sintácticos y cotas de variables

Se definen los predicados inductivos y funciones que permiten razonar sobre la ocurrencia y no ocurrencia de variables en términos y fórmulas, así como cotas superiores para las variables que aparecen en una expresión. Estas herramientas son fundamentales para garantizar la frescura de variables durante las operaciones de sustitución y cuantificación.

Predicado inductivo `NotOccur`: No ocurrencia de una variable en un término

Define inductivamente cuándo una variable con índice n no ocurre bajo ninguna circunstancia dentro de un término aritmético t:

$$ \text{NotOccur } n\ t : \text{Prop} $$

Las reglas de construcción cubren todos los constructores de `ArithTerm`:

- `var m (h : n ≠ m)`: La variable n no ocurre en `var m` si $n \neq m$.

- `const c`, `henkin c`, `zero`: Las constantes, Henkin y cero no contienen variables, por lo que n no ocurre en ellos.

- `succ t (h)`, `pred t (h)`: Si n no ocurre en t, tampoco ocurre en $S(t)$ ni en $\text{pred}(t)$.

- `add t1 t2 (h1, h2)`, `mul`, `exp`: Si n no ocurre en ninguno de los subtérminos, tampoco ocurre en el término compuesto.

Intuición y necesidad

En el análisis sintáctico de términos, es fundamental identificar si una variable libre interviene o no en la expresión de un término para evitar capturas durante reescrituras o generalizaciones lógicas. Este predicado proporciona una caracterización inductiva que permite demostrar propiedades por inversión sobre la estructura del término.

Función `NotOccurForm`: No ocurrencia de una variable en una fórmula

Extiende recursivamente el concepto de no ocurrencia sintáctica de una variable v a las proposiciones del tipo `ArithForm s`:

$$ \text{NotOccurForm } v\ \varphi : \text{Prop} $$

La definición por casos sobre la estructura de φ establece:

- `bottom`: Siempre verdadero (no hay variables).

- `eq t1 t2`: v no ocurre en t_1 ni en t_2.

- `impl f1 f2`, `neg f`: Se propaga recursivamente a las subfórmulas.

- `forall_form n f`: $v \neq n$ (la variable ligada es distinta) y v no ocurre en f.

- `provable_op`, `true_op`: v no ocurre en el término argumento.

- `list_nth_eq`: v no ocurre en ninguno de los tres términos.

Intuición y necesidad

Es la herramienta sintáctica clave para formalizar que una variable no aparece en absoluto (ni libre ni ligada) en una fórmula φ. Esta noción estricta de no ocurrencia se requiere para validar las condiciones de aplicabilidad de la regla de generalización universal ($\forall$-intro) y para garantizar la corrección de las operaciones de sustitución.

Función `max_var_term`: Máximo índice de variable en un término

Calcula el mayor índice de variable que aparece en un término:

$$ \text{max_var_term} : \text{ArithTerm } s \to \mathbb{N} $$

Para `var n` devuelve n; para constantes, Henkin y cero devuelve 0; para constructores unarios propaga el máximo del subtérmino; para constructores binarios toma el máximo de ambos subtérminos.

Lema `NotOccur_above_max_var_term`: Frescura por cota superior

Establece que cualquier variable cuyo índice es estrictamente mayor que el máximo índice de variable en un término no puede ocurrir en dicho término:

$$ \forall t \in \text{ArithTerm } s,\ \forall w \in \mathbb{N},\ w > \text{max_var_term}(t) \implies \text{NotOccur } w\ t $$

Estructura de la demostración

La prueba procede por inducción estructural sobre t:

- **Caso `var n`:** Si $w > n$, entonces $w \neq n$ y se aplica el constructor `NotOccur.var`.

- **Casos base (`const`, `henkin`, `zero`):** Son inmediatos pues estos constructores no contienen variables.

- **Casos unarios (`succ`, `pred`):** Se aplica la hipótesis de inducción, ya que el máximo no cambia.

- **Casos binarios (`add`, `mul`, `exp`):** El máximo de los subtérminos es una cota superior para cada uno individualmente (por `Nat.le_max_left` y `Nat.le_max_right`). Si w supera esta cota, supera también la de cada subtérmino, y las hipótesis de inducción proporcionan `NotOccur w` para ambos.

Significado

Este lema es una herramienta fundamental para generar variables frescas: dado un término t, cualquier variable con índice mayor que `max_var_term(t)` está garantizada como no ocurriente en t. Esta propiedad se utiliza extensivamente en las demostraciones por inducción interna (como las de `add_comm_Δ`, `add_assoc_Δ_term`, `succ_mul_Δ`, etc.) donde se necesita elegir una variable de inducción que no aparezca en los términos involucrados para evitar capturas. La existencia de un suministro ilimitado de variables frescas (eligiendo índices suficientemente grandes) es un requisito indispensable para la manipulación sintáctica en sistemas formales con ligaduras explícitas.

--
==

-- PREDICADOS SINTÁCTICOS Y COTAS DE VARIABLES

--
==

```
inductive NotOccur (n : Nat) {s : Nat} : ArithTerm s → Prop where
  | var (m : Nat) (h : n ≠ m) : NotOccur n (ArithTerm.var m)
  | const (c : NatS s) : NotOccur n (ArithTerm.const c)
  | henkin (c : HenkinConst) : NotOccur n (ArithTerm.henkin c)
  | zero : NotOccur n ArithTerm.zero
  | succ (t : ArithTerm s) (h : NotOccur n t) : NotOccur n (ArithTerm.succ t)
  | add (t1 t2 : ArithTerm s) (h1 : NotOccur n t1) (h2 : NotOccur n t2) : NotOccur n (ArithTerm.add t1 t2)
  | mul (t1 t2 : ArithTerm s) (h1 : NotOccur n t1) (h2 : NotOccur n t2) : NotOccur n (ArithTerm.mul t1 t2)
  | exp (t1 t2 : ArithTerm s) (h1 : NotOccur n t1) (h2 : NotOccur n t2) : NotOccur n (ArithTerm.exp t1 t2)
  | pred (t : ArithTerm s) (h : NotOccur n t) : NotOccur n (ArithTerm.pred t)

def NotOccurForm {s : Nat} (v : Nat) : ArithForm s → Prop
  | ArithForm.bottom => True
  | ArithForm.eq t1 t2 => NotOccur v t1 ∧ NotOccur v t2
  | ArithForm.impl f1 f2 => NotOccurForm v f1 ∧ NotOccurForm v f2
  | ArithForm.neg f => NotOccurForm v f
  | ArithForm.forall_form n f => v ≠ n ∧ NotOccurForm v f
  | ArithForm.provable_op _ t => NotOccur v t
  | ArithForm.true_op _ t => NotOccur v t
  | ArithForm.list_nth_eq t1 t2 t3 => NotOccur v t1 ∧ NotOccur v t2 ∧ NotOccur v t3

def max_var_term {s : Nat} : ArithTerm s → Nat
  | ArithTerm.var n => n
  | ArithTerm.const _ => 0
  | ArithTerm.henkin _ => 0
```

```
  | ArithTerm.zero => 0
  | ArithTerm.succ t => max_var_term t
  | ArithTerm.add t1 t2 => Nat.max (max_var_term t1) (max_var_term t2)
  | ArithTerm.mul t1 t2 => Nat.max (max_var_term t1) (max_var_term t2)
  | ArithTerm.exp t1 t2 => Nat.max (max_var_term t1) (max_var_term t2)
  | ArithTerm.pred t => max_var_term t

lemma NotOccur_above_max_var_term {s : Nat} (t : ArithTerm s) (w : Nat) (h : w > max_var_term t) : NotOccur w t := by
  induction t with
  | var n => unfold max_var_term at h; apply NotOccur.var; omega
  | const c => exact NotOccur.const _
  | henkin c => exact NotOccur.henkin _
  | zero => exact NotOccur.zero
  | succ t' ih => unfold max_var_term at h; apply NotOccur.succ; apply ih h
  | add t1 t2 ih1 ih2 =>
    unfold max_var_term at h
    have h1 : max_var_term t1 ≤ Nat.max (max_var_term t1) (max_var_term t2) := Nat.le_max_left _ _
    have h2 : max_var_term t2 ≤ Nat.max (max_var_term t1) (max_var_term t2) := Nat.le_max_right _ _
    exact NotOccur.add _ _ (ih1 (by omega)) (ih2 (by omega))
  | mul t1 t2 ih1 ih2 =>
    unfold max_var_term at h
    have h1 : max_var_term t1 ≤ Nat.max (max_var_term t1) (max_var_term t2) := Nat.le_max_left _ _
    have h2 : max_var_term t2 ≤ Nat.max (max_var_term t1) (max_var_term t2) := Nat.le_max_right _ _
    exact NotOccur.mul _ _ (ih1 (by omega)) (ih2 (by omega))
  | exp t1 t2 ih1 ih2 =>
```

```
  unfold max_var_term at h
  have h1 : max_var_term t1 ≤ Nat.max (max_var_term t1) (max_var_term t2) := Nat.le_max_left _ _
  have h2 : max_var_term t2 ≤ Nat.max (max_var_term t1) (max_var_term t2) := Nat.le_max_right _ _
  exact NotOccur.exp _ _ (ih1 (by omega)) (ih2 (by omega))
| pred t' ih => unfold max_var_term at h; apply NotOccur.pred; apply ih h
```

Funciones de máximo de variables en fórmulas y listas, y lema de frescura

Se completan las definiciones de cotas superiores para variables en fórmulas y listas de fórmulas, junto con el lema fundamental que garantiza frescura por encima de dichas cotas.

Función `max_var_form`: Máximo índice de variable en una fórmula

Calcula el mayor índice de variable que aparece en una fórmula (libre o ligada):

$$ \text{max_var_form} : \text{ArithForm } s \to \mathbb{N} $$

La definición es recursiva sobre la estructura de la fórmula:

- `bottom`: 0 (no contiene variables).

- `eq t1 t2`: El máximo entre los máximos de t_1 y t_2.

- `impl f1 f2`, `forall_form n f`: El máximo de los máximos de las subfórmulas. En `forall_form`, se incluye también n (la variable ligada) en el cálculo del máximo.

- `neg f`: Se propaga a la subfórmula.

- `provable_op`, `true_op`: El máximo del término argumento.

- `list_nth_eq t1 t2 t3`: El máximo de los tres términos.

Intuición y necesidad

Esta función proporciona una cota superior para todas las variables que aparecen en una fórmula, ya sea libre o ligadas. Es una herramienta esencial para generar variables frescas respecto a una fórmula completa, necesarias en operaciones como la introducción de cuantificadores o la instanciación de testigos de Henkin.

Función `max_var_list`: Máximo índice de variable en una lista de fórmulas

Extiende el concepto a listas de fórmulas, calculando el máximo de los máximos individuales:

$$ \text{max_var_list} : \text{List} (\text{ArithForm } s) \to \mathbb{N} $$

$$ \text{max_var_list}([]) = 0 $$

$$ \text{max_var_list}(f :: fs) = \max(\text{max_var_form}(f), \text{max_var_list}(fs)) $$

Intuición y necesidad

Durante la construcción de la secuencia de Henkin, se mantiene una lista de fórmulas testigo acumuladas. Esta función permite calcular una cota superior para todas las variables que han aparecido hasta el momento, garantizando que las nuevas variables y constantes de Henkin elegidas sean estrictamente mayores y por tanto frescas.

Lema `NotOccurForm_of_gt_max_var`: Frescura en fórmulas por cota superior

Generaliza `NotOccur_above_max_var_term` al nivel de fórmulas: cualquier variable cuyo índice es estrictamente mayor que `max_var_form(φ)` no puede ocurrir en φ (ni libre ni ligada):

$$ \forall \varphi \in \text{ArithForm } s,\ \forall w \in \mathbb{N},\ w > \text{max_var_form}(\varphi) \implies \text{NotOccurForm } w\ \varphi $$

Estructura de la demostración

La prueba procede por inducción estructural sobre φ:

- **`bottom`:** Trivial, pues no hay variables.

- **`eq t1 t2`:** Dado que w supera el máximo de ambos términos, por `NotOccur_above_max_var_term` no ocurre en t_1 ni en t_2. La conjunción de ambas condiciones da `NotOccurForm w`.

- **`impl f1 f2`:** Similar, usando las hipótesis de inducción para f_1 y f_2.

- **`neg f`:** Directo por hipótesis de inducción.

- **`forall_form n f`:** w supera tanto a n como a `max_var_form(f)`. Por tanto $w \neq n$, y por hipótesis de inducción w no ocurre en f. La conjunción de ambas establece `NotOccurForm w`.

- **`provable_op`, `true_op`:** Se reduce a `NotOccur_above_max_var_term`.

- **`list_nth_eq t1 t2 t3`:** w supera el máximo de los tres términos individualmente, por lo que no ocurre en ninguno de ellos. Se requiere descomponer el máximo anidado usando las propiedades `Nat.le_max_left` y `Nat.le_max_right` dos veces.

Significado

`NotOccurForm_of_gt_max_var` es el análogo para fórmulas de `NotOccur_above_max_var_term`. Junto con `max_var_list`, proporciona un mecanismo sistemático para generar variables y constantes de Henkin frescas durante la construcción de Lindenbaum: basta elegir un índice mayor que el máximo de todas las variables aparecidas hasta el momento. Esta técnica de "frescura por cota superior" es ubicua en las demostraciones de inducción interna y en la manipulación de testigos existenciales a lo largo de todo el desarrollo del sistema SPA. La simplicidad del enunciado oculta su importancia: sin una fuente garantizada de símbolos frescos, todas las construcciones que involucran cuantificadores y sustituciones serían potencialmente incorrectas debido a capturas de variables.

```
def max_var_form {s : Nat} : ArithForm s → Nat
  | ArithForm.bottom => 0
  | ArithForm.eq t1 t2 => Nat.max (max_var_term t1) (max_var_term t2)
```

```
 | ArithForm.impl f1 f2 => Nat.max (max_var_form f1) (max_var_form f2)
 | ArithForm.neg f => max_var_form f
 | ArithForm.forall_form n f => Nat.max n (max_var_form f)
 | ArithForm.provable_op _ t => max_var_term t
 | ArithForm.true_op _ t => max_var_term t
 | ArithForm.list_nth_eq t1 t2 t3 => Nat.max (max_var_term t1) (Nat.max (max_var_term t2) (max_var_term t3))

def max_var_list {s : Nat} : List (ArithForm s) → Nat
 | [] => 0
 | f :: fs => Nat.max (max_var_form f) (max_var_list fs)

lemma NotOccurForm_of_gt_max_var {s : Nat} (φ : ArithForm s) (w : Nat) (h : w > max_var_form φ) : NotOccurForm w φ := by
 induction φ with
 | bottom => unfold max_var_form at h; unfold NotOccurForm; trivial
 | eq t1 t2 =>
  unfold max_var_form at h; unfold NotOccurForm
  have h1 : max_var_term t1 ≤ Nat.max (max_var_term t1) (max_var_term t2) := Nat.le_max_left _ _
  have h2 : max_var_term t2 ≤ Nat.max (max_var_term t1) (max_var_term t2) := Nat.le_max_right _ _
  exact ⟨NotOccur_above_max_var_term t1 w (by omega), NotOccur_above_max_var_term t2 w (by omega)⟩
 | impl f1 f2 ih1 ih2 =>
  unfold max_var_form at h; unfold NotOccurForm
  have h1 : max_var_form f1 ≤ Nat.max (max_var_form f1) (max_var_form f2) := Nat.le_max_left _ _
  have h2 : max_var_form f2 ≤ Nat.max (max_var_form f1) (max_var_form f2) := Nat.le_max_right _ _
  exact ⟨ih1 (by omega), ih2 (by omega)⟩
```

```
  | neg f ih => unfold max_var_form at h; unfold NotOccurForm; exact ih h
  | forall_form n f ih =>
    unfold max_var_form at h; unfold NotOccurForm
    have h1 : n ≤ Nat.max n (max_var_form f) := Nat.le_max_left _ _
    have h2 : max_var_form f ≤ Nat.max n (max_var_form f) := Nat.le_max_right _ _
    exact ⟨by omega, ih (by omega)⟩
  | provable_op _ t => unfold max_var_form at h; unfold NotOccurForm; exact NotOccur_above_max_var_term t w h
  | true_op _ t => unfold max_var_form at h; unfold NotOccurForm; exact NotOccur_above_max_var_term t w h
  | list_nth_eq t1 t2 t3 =>
    unfold max_var_form at h; unfold NotOccurForm
    have h1 : max_var_term t1 ≤ Nat.max (max_var_term t1) (Nat.max (max_var_term t2) (max_var_term t3)) := Nat.le_max_left _ _
    have h2 : Nat.max (max_var_term t2) (max_var_term t3) ≤ Nat.max (max_var_term t1) (Nat.max (max_var_term t2) (max_var_term t3)) := Nat.le_max_right _ _
    have h3 : max_var_term t2 ≤ Nat.max (max_var_term t2) (max_var_term t3) := Nat.le_max_left _ _
    have h4 : max_var_term t3 ≤ Nat.max (max_var_term t2) (max_var_term t3) := Nat.le_max_right _ _
    exact ⟨NotOccur_above_max_var_term t1 w (by omega), NotOccur_above_max_var_term t2 w (by omega), NotOccur_above_max_var_term t3 w (by omega)⟩
```

Lema de frescura para listas y funciones de sustitución sintáctica

Se completa la infraestructura sintáctica con un lema de frescura para listas de fórmulas, las definiciones de sustitución para términos y fórmulas, y el predicado que garantiza que una sustitución es segura (no causa captura de variables).

Lema `NotOccurForm_list_of_gt_max_list`: Frescura uniforme en listas

Establece que si una variable w es estrictamente mayor que el máximo de variables de una lista de fórmulas L, entonces w no ocurre en ninguna fórmula de la lista:

$$ \forall L \in \text{List}(\text{ArithForm } s),\ \forall w > \text{max_var_list}(L),\ \forall \varphi \in L,\ \text{NotOccurForm } w\ \varphi $$

Estructura de la demostración

Por inducción sobre L:

- **Lista vacía:** No hay fórmulas, la conclusión es vacuamente cierta.

- **Lista con cabeza f y cola fs:** Dado que w supera el máximo de la lista completa, en particular supera `max_var_form(f)` y `max_var_list(fs)`. Para una fórmula φ en la lista, o bien es f (y se aplica `NotOccurForm_of_gt_max_var`), o bien está en fs (y se aplica la hipótesis de inducción).

Función `substTerm`: Sustitución de una variable por un término en un término

Define la operación de sustitución sintáctica de la variable v por el término t dentro de un término:

$$ \text{substTerm}(v, t, \cdot) : \text{ArithTerm } s \to \text{ArithTerm } s $$

La definición es recursiva estructural:

- `var n`: Si $n = v$, se reemplaza por t; en caso contrario, se preserva.
- `const`, `henkin`, `zero`: Inalterados (no contienen variables).
- `succ`, `add`, `mul`, `exp`, `pred`: Se propagan recursivamente a los subtérminos.

Función `substForm`: Sustitución de una variable por un término en una fórmula

Extiende la sustitución al nivel de fórmulas:

$$ \text{substForm}(v, t, \cdot) : \text{ArithForm } s \to \text{ArithForm } s $$

- `bottom`: Inalterado.
- `eq t1 t2`, `impl f1 f2`, `neg f`: Se propagan recursivamente a subtérminos y subfórmulas.
- **`forall_form n f`:** Si $v = n$, la variable está ligada y la sustitución no afecta a la fórmula. Si $v \neq n$, se sustituye en el cuerpo f. Esta condición es crucial para evitar la captura de variables ligadas.
- `provable_op`, `true_op`, `list_nth_eq`: Se propagan a los subtérminos.

Predicado `IsFreeFor`: Condición de sustitución segura

Define cuándo un término t es libre para la variable v en una fórmula φ, es decir, la sustitución de v por t en φ no causa captura de variables libres de t por cuantificadores de φ:

$$ \text{IsFreeFor}(v, t, \varphi) : \text{Prop} $$

- Para `bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`: Siempre verdadero (no hay cuantificadores que puedan capturar).

- Para `impl f1 f2`, `neg f`: Se propaga recursivamente.

- **Para `forall_form n f`:** Si $v = n$, la sustitución no penetra el cuantificador, por lo que es segura. Si $v \neq n$, se requiere que n no ocurra en t (para evitar que una variable libre de t sea capturada por el cuantificador) y que t sea libre para v en f.

Lemas `IsFreeFor_provable_op` e `IsFreeFor_true_op`

Establecen que para los operadores estratificados, la condición `IsFreeFor` es siempre verdadera, ya que estos constructores no contienen cuantificadores que puedan capturar variables:

$$ \text{IsFreeFor}(v, t, \text{provable_op } h\ t') = \text{True} $$

$$ \text{IsFreeFor}(v, t, \text{true_op } h\ t') = \text{True} $$

Significado

Estas definiciones constituyen el núcleo de la manipulación sintáctica en el sistema SPA. La sustitución es la operación fundamental para la instanciación de cuantificadores y la construcción de fórmulas autorreferentes. El predicado

`IsFreeFor` captura la condición técnica necesaria para que la regla de eliminación del cuantificador universal ($\forall$-elim) sea aplicable: el término que se sustituye no debe tener variables que resulten capturadas por cuantificadores en el alcance de la sustitución. La distinción en `substForm` entre $v = n$ y $v \neq n$ para `forall_form` implementa la convención estándar de que las variables ligadas no son sustituibles, mientras que `IsFreeFor` refina esta condición exigiendo además que no haya captura cuando la variable de sustitución es distinta de la ligada. Estas definiciones son la base sobre la cual se construye todo el sistema deductivo `SPA_Deduction_Internal`.

```
lemma NotOccurForm_list_of_gt_max_list {s : Nat} (L : List (ArithForm s)) (w : Nat) (h : w > max_var_list L) : ∀ φ ∈ L, NotOccurForm w φ := by

 induction L with

 | nil => intro φ h_mem; nomatch h_mem

 | cons f fs ih =>

  unfold max_var_list at h

  have h1 : max_var_form f ≤ Nat.max (max_var_form f) (max_var_list fs) := Nat.le_max_left _ _

  have h2 : max_var_list fs ≤ Nat.max (max_var_form f) (max_var_list fs) := Nat.le_max_right _ _

  intro φ h_mem

  rw [List.mem_cons] at h_mem

  cases h_mem with

  | inl h_eq => rw [h_eq]; exact NotOccurForm_of_gt_max_var f w (by omega)
```

```
  | inr h_mem_fs => exact ih (by omega) φ h_mem_fs

def substTerm {s : Nat} (v : Nat) (t : ArithTerm s) : ArithTerm s → ArithTerm s
  | ArithTerm.var n => if v = n then t else ArithTerm.var n
  | ArithTerm.const c => ArithTerm.const c
  | ArithTerm.henkin c => ArithTerm.henkin c
  | ArithTerm.zero => ArithTerm.zero
  | ArithTerm.succ t' => ArithTerm.succ (substTerm v t t')
  | ArithTerm.add t1 t2 => ArithTerm.add (substTerm v t t1) (substTerm v t t2)
  | ArithTerm.mul t1 t2 => ArithTerm.mul (substTerm v t t1) (substTerm v t t2)
  | ArithTerm.exp t1 t2 => ArithTerm.exp (substTerm v t t1) (substTerm v t t2)
  | ArithTerm.pred t' => ArithTerm.pred (substTerm v t t')

def substForm {s : Nat} (v : Nat) (t : ArithTerm s) : ArithForm s → ArithForm s
  | ArithForm.bottom => ArithForm.bottom
  | ArithForm.eq t1 t2 => ArithForm.eq (substTerm v t t1) (substTerm v t t2)
  | ArithForm.impl f1 f2 => ArithForm.impl (substForm v t f1) (substForm v t f2)
  | ArithForm.neg f => ArithForm.neg (substForm v t f)
  | ArithForm.forall_form n f => if v = n then ArithForm.forall_form n f else
ArithForm.forall_form n (substForm v t f)
  | ArithForm.provable_op h t' => ArithForm.provable_op h (substTerm v t t')
  | ArithForm.true_op h t' => ArithForm.true_op h (substTerm v t t')
  | ArithForm.list_nth_eq t1 t2 t3 => ArithForm.list_nth_eq (substTerm v t t1)
(substTerm v t t2) (substTerm v t t3)

def IsFreeFor {s : Nat} (v : Nat) (t : ArithTerm s) : ArithForm s → Prop
  | ArithForm.bottom => True
  | ArithForm.eq _ _ => True
  | ArithForm.impl f1 f2 => IsFreeFor v t f1 ∧ IsFreeFor v t f2
```

```
  | ArithForm.neg f => IsFreeFor v t f
  | ArithForm.forall_form n f => if v = n then True else NotOccur n t ∧ IsFreeFor v t f
  | ArithForm.provable_op _ _ => True
  | ArithForm.true_op _ _ => True
  | ArithForm.list_nth_eq _ _ _ => True
```

```
lemma IsFreeFor_provable_op {s : Nat} (v : Nat) (t : ArithTerm s) {s_lower : Nat} (_h : s
= s_lower + 1) (t' : ArithTerm s) :
    IsFreeFor v t (ArithForm.provable_op _h t') = True := rfl
```

```
lemma IsFreeFor_true_op {s : Nat} (v : Nat) (t : ArithTerm s) {s_lower : Nat} (_h : s =
s_lower + 1) (t' : ArithTerm s) :
    IsFreeFor v t (ArithForm.true_op _h t') = True := rfl
```

Predicados de variables libres, conjuntos libres y variables ligadas

Se definen tres predicados sintácticos fundamentales para el manejo de variables en el sistema deductivo, junto con un lema que relaciona la condición de variable ligada con la seguridad de sustitución.

Predicado `NotFreeForm`: Variable no libre en una fórmula

Define cuándo una variable v no ocurre libre en una fórmula φ. A diferencia de `NotOccurForm` (que exige que la variable no aparezca en absoluto), este predicado solo requiere que no aparezca libre, pudiendo aparecer ligada:

$$ \text{NotFreeForm } v\ \varphi : \text{Prop} $$

La definición recursiva sobre φ establece:

- `bottom`: Siempre verdadero.

- `eq t1 t2`, `list_nth_eq`: La variable no debe ocurrir en los términos (si aparece en un término, es libre).

- `impl f1 f2`, `neg f`: Se propaga recursivamente a las subfórmulas.

- **`forall_form n f`:** Si $v = n$, la variable está ligada por el cuantificador, por lo que no es libre (la condición se cumple trivialmente). Si $v \neq n$, se requiere que v no sea libre en f.

- `provable_op`, `true_op`: La variable no debe ocurrir en el término argumento.

Intuición y necesidad

Este predicado es la contraparte formal de la noción estándar de "variable ligada" en lógica de primer orden. Es esencial para la regla de generalización universal ($\forall$-intro), que requiere que la variable generalizada no ocurra libre en las hipótesis. También es utilizado en la definición de `IsRepresentable`, donde se exige que la fórmula representadora tenga exactamente dos variables libres (v_0 y v_1), lo cual se expresa mediante $\neg\text{NotFreeForm }0\ \varphi$ y $\neg\text{NotFreeForm }1\ \varphi$, junto con $\text{NotFreeForm } v\ \varphi$ para todo $v \neq 0,1$.

Predicado `NotFreeInSet`: Variable no libre en un conjunto de fórmulas

Extiende `NotFreeForm` a conjuntos de fórmulas, exigiendo que v no sea libre en ninguna fórmula del conjunto:

$$ \text{NotFreeInSet } v\ \Gamma := \forall \psi \in \Gamma,\ \text{NotFreeForm } v\ \psi $$

Intuición y necesidad

Esta noción es necesaria para la regla de generalización universal en contextos con hipótesis: para introducir $\forall v\ \varphi$, se requiere que v no ocurra libre en ninguna fórmula del conjunto de hipótesis Γ. `NotFreeInSet` captura precisamente esta condición.

Predicado `NoBoundVar`: Variable no ligada en una fórmula

Define cuándo una variable v no aparece ligada en una fórmula φ (aunque podría aparecer libre):

$$ \text{NoBoundVar } v\ \varphi : \text{Prop} $$

- Para casos base (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`): Siempre verdadero.

- Para `impl f1 f2`, `neg f`: Se propaga recursivamente.

- **Para `forall_form n f`:** Se requiere $v \neq n$ (la variable ligada es distinta de v) y que v no esté ligada en f.

Intuición y necesidad

Este predicado se utiliza en los lemas de cambio de variable ligada (alpha-conversión). Cuando se necesita renombrar una variable ligada para evitar capturas, `NoBoundVar`

garantiza que la nueva variable elegida no aparece ya como ligada en la fórmula, evitando conflictos. Es la condición técnica que permite aplicar el lema `change_var_equivalence_strong`.

Lema `IsFreeFor_var`: Una variable ligada es libre para cualquier sustitución

Establece que si una variable w no aparece ligada en una fórmula φ, entonces el término `var w` es libre para cualquier variable v en φ:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \text{NoBoundVar } w\ \varphi \implies \text{IsFreeFor } v\ (\text{var } w)\ \varphi $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos base (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`):** `IsFreeFor` es siempre verdadero para estos constructores.

- **`impl f1 f2`:** Se aplican las hipótesis de inducción a ambas subfórmulas.

- **`neg f`:** Se aplica la hipótesis de inducción.

- **`forall_form n f`:** Se distingue si $v = n$:

  - Si $v = n$, la sustitución no penetra el cuantificador, y `IsFreeFor` es verdadero.

  - Si $v \neq n$, la hipótesis `NoBoundVar w (forall_form n f)` proporciona $w \neq n$ y `NoBoundVar w f`. Se debe demostrar `NotOccur n (var w)` (que se cumple porque $w \neq n$) y `IsFreeFor v (var w) f` (que se cumple por hipótesis de inducción).

Significado

Este lema es técnicamente importante porque establece que las variables que no aparecen ligadas son "inocuas" para la sustitución: pueden ser sustituidas sin riesgo de captura. En particular, garantiza que al aplicar la regla de eliminación del universal ($\forall$-elim) instanciando con una variable fresca, la condición `IsFreeFor` se satisface automáticamente. Esto es utilizado extensivamente en las demostraciones de inducción interna, donde se instancian cuantificadores universales con la variable de inducción (que se elige fresca precisamente para que no esté ligada en la fórmula).

```
def NotFreeForm {s : Nat} (v : Nat) : ArithForm s → Prop
  | ArithForm.bottom => True
  | ArithForm.eq t1 t2 => NotOccur v t1 ∧ NotOccur v t2
  | ArithForm.impl f1 f2 => NotFreeForm v f1 ∧ NotFreeForm v f2
  | ArithForm.neg f => NotFreeForm v f
  | ArithForm.forall_form n f => if v = n then True else NotFreeForm v f
  | ArithForm.provable_op _ t => NotOccur v t
  | ArithForm.true_op _ t => NotOccur v t
  | ArithForm.list_nth_eq t1 t2 t3 => NotOccur v t1 ∧ NotOccur v t2 ∧ NotOccur v t3

def NotFreeInSet {s : Nat} (v : Nat) (Γ : Set (ArithForm s)) : Prop :=
  ∀ ψ ∈ Γ, NotFreeForm v ψ

def NoBoundVar {s : Nat} (v : Nat) : ArithForm s → Prop
  | ArithForm.bottom => True
```

```
  | ArithForm.eq _ _ => True
  | ArithForm.impl f1 f2 => NoBoundVar v f1 ∧ NoBoundVar v f2
  | ArithForm.neg f => NoBoundVar v f
  | ArithForm.forall_form n f => v ≠ n ∧ NoBoundVar v f
  | ArithForm.provable_op _ _ => True
  | ArithForm.true_op _ _ => True
  | ArithForm.list_nth_eq _ _ _ => True

lemma IsFreeFor_var {s : Nat} (φ : ArithForm s) (v w : Nat) (h_bound : NoBoundVar w
φ) :
    IsFreeFor v (ArithTerm.var w) φ := by
  induction φ generalizing v w with
  | bottom => trivial
  | eq _ _ => trivial
  | impl f1 f2 ih1 ih2 =>
    dsimp [IsFreeFor]
    exact ⟨ih1 v w h_bound.1, ih2 v w h_bound.2⟩
  | neg f ih =>
    dsimp [IsFreeFor]
    exact ih v w h_bound
  | forall_form n f ih =>
    dsimp [IsFreeFor]
    by_cases h_vn : v = n
```

```
    · rw [if_pos h_vn]
      trivial
    · rw [if_neg h_vn]
      constructor
      · exact NotOccur.var w (Ne.symm h_bound.1)
      · exact ih v w h_bound.2
  | provable_op _ _ => trivial
  | true_op _ _ => trivial
  | list_nth_eq _ _ _ => trivial
```

Sistema deductivo `SPA_Deduction_Internal`

Se define el sistema deductivo para la aritmética estratificada SPA como un predicado inductivo en `Prop`. Este sistema constituye el núcleo lógico sobre el cual se desarrolla toda la teoría de la demostración y la representabilidad.

Definición inductiva

$$ \text{SPA_Deduction_Internal } s\ \Gamma\ \varphi : \text{Prop} $$

representa que la fórmula φ es deducible del conjunto de hipótesis Γ en el estrato s. Las reglas de inferencia son las siguientes:

Reglas lógicas básicas

- **`hyp`:** Si $\varphi \in \Gamma$, entonces $\Gamma \vdash \varphi$ (axioma de hipótesis).

- **`imp_intro`:** Si $\Gamma \cup \{\varphi\} \vdash \psi$, entonces $\Gamma \vdash \varphi \to \psi$ (teorema de la deducción).

- **`imp_elim`:** Si $\Gamma \vdash \varphi \to \psi$ y $\Gamma \vdash \varphi$, entonces $\Gamma \vdash \psi$ (modus ponens).

- **`double_neg`:** Si $\Gamma \vdash \neg\neg\varphi$, entonces $\Gamma \vdash \varphi$ (eliminación de la doble negación).

- **`neg_intro`:** Si $\Gamma \cup \{\varphi\} \vdash \bot$, entonces $\Gamma \vdash \neg\varphi$ (reducción al absurdo).

- **`neg_elim`:** Si $\Gamma \vdash \varphi$ y $\Gamma \vdash \neg\varphi$, entonces $\Gamma \vdash \bot$ (principio de no contradicción).

Reglas para cuantificadores

- **`forall_intro`:** Si $\Gamma \vdash \varphi$ y v no es libre en Γ, entonces $\Gamma \vdash \forall v\ \varphi$ (generalización universal). La

condición `NotFreeInSet v Γ` es la restricción estándar que evita la generalización sobre variables que aparecen libres en las hipótesis.

- **`forall_elim`:** Si $\Gamma \vdash \forall v\ \varphi$ y t es libre para v en φ, entonces $\Gamma \vdash \varphi[v := t]$ (instanciación universal). La condición `IsFreeFor v t φ` garantiza que la sustitución no causa captura de variables.

Reglas de igualdad

- **`eq_refl`:** $\Gamma \vdash t = t$ (reflexividad).

- **`eq_symm`:** Si $\Gamma \vdash t_1 = t_2$, entonces $\Gamma \vdash t_2 = t_1$ (simetría).

- **`eq_trans`:** Si $\Gamma \vdash t_1 = t_2$ y $\Gamma \vdash t_2 = t_3$, entonces $\Gamma \vdash t_1 = t_3$ (transitividad).

- **`eq_succ`, `eq_add`, `eq_mul`, `eq_exp`, `eq_pred`:** Reglas de congruencia para cada constructor de términos: si dos términos son iguales, sus imágenes bajo el constructor también lo son.

- **`eq_provable_op`, `eq_true_op`:** Congruencia para los operadores estratificados, requiriendo también la premisa adicional de que la fórmula original es demostrable.

- **`eq_list_nth_eq_1/2/3`:** Congruencia para cada argumento de `list_nth_eq`.

- **`eq_subst`:** Si $\Gamma \vdash t_1 = t_2$ y $\Gamma \vdash \varphi[v := t_1]$, y ambas sustituciones son seguras, entonces $\Gamma \vdash \varphi[v := t_2]$ (sustitutividad de la igualdad).

Reglas estructurales y aritméticas

- **`weaken`:** Si $\Gamma \vdash \varphi$ y $\Gamma \subseteq \Gamma'$, entonces $\Gamma' \vdash \varphi$ (monotonía).

- **`ax_peano1`:** $\Gamma \vdash \neg(S(x) = 0)$ para cualquier término x (el cero no es sucesor).

- **`ax_peano2`:** $\Gamma \vdash S(x) = S(y) \to x = y$ (inyectividad del sucesor).

- **`induction`:** Esquema de inducción: si $\Gamma \vdash \varphi[0]$ (caso base) y $\Gamma \vdash \forall v\ (\varphi[v] \to \varphi[S(v)])$ (paso inductivo), entonces $\Gamma \vdash \forall v\ \varphi[v]$.

Significado global

`SPA_Deduction_Internal` es un sistema deductivo completo para la lógica de primer orden con igualdad, enriquecido con los axiomas de Peano (sucesor inyectivo y distinto de cero) y el esquema de inducción. La estratificación se refleja en el parámetro s y en la presencia de reglas específicas para los operadores `provable_op` y `true_op`.

Este sistema es la base sobre la cual se define la noción de consistencia, maximalidad y propiedad de Henkin para las teorías Δ. Todas las demostraciones de representabilidad y los lemas algebraicos (como `add_comm_Δ`, `succ_mul_Δ`, `mul_comm_Δ`) utilizan estas reglas como bloques de construcción elementales. La regla `induction` es particularmente importante, ya que permite demostrar propiedades universales sobre los términos mediante el principio de inducción matemática internalizado en el lenguaje objeto. Las reglas de congruencia para los operadores estratificados (`eq_provable_op`, `eq_true_op`) reflejan el carácter intensional de estos operadores: la igualdad de los argumentos no es suficiente para deducir la igualdad de las fórmulas compuestas; se requiere además que la fórmula original sea demostrable.

--
==

-- SISTEMA DEDUCTIVO Y DERIVABILIDAD (en Prop)

--
==

inductive SPA_Deduction_Internal (s : Nat) : Set (ArithForm s) → ArithForm s → Prop where

| hyp (Γ : Set (ArithForm s)) (φ : ArithForm s) : φ ∈ Γ → SPA_Deduction_Internal s Γ φ

| imp_intro (Γ : Set (ArithForm s)) (φ ψ : ArithForm s) :

SPA_Deduction_Internal s (insert φ Γ) ψ → SPA_Deduction_Internal s Γ (ArithForm.impl φ ψ)

| imp_elim (Γ : Set (ArithForm s)) (φ ψ : ArithForm s) :

SPA_Deduction_Internal s Γ (ArithForm.impl φ ψ) → SPA_Deduction_Internal s Γ φ → SPA_Deduction_Internal s Γ ψ

| double_neg (Γ : Set (ArithForm s)) (φ : ArithForm s) :

SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.neg φ)) → SPA_Deduction_Internal s Γ φ

| neg_intro (Γ : Set (ArithForm s)) (φ : ArithForm s) :

SPA_Deduction_Internal s (insert φ Γ) ArithForm.bottom → SPA_Deduction_Internal s Γ (ArithForm.neg φ)

| neg_elim (Γ : Set (ArithForm s)) (φ : ArithForm s) :

SPA_Deduction_Internal s Γ φ → SPA_Deduction_Internal s Γ (ArithForm.neg φ) → SPA_Deduction_Internal s Γ ArithForm.bottom

| forall_intro (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s) :

SPA_Deduction_Internal s Γ φ → NotFreeInSet v Γ → SPA_Deduction_Internal s Γ (ArithForm.forall_form v φ)

| forall_elim (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s) (t : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.forall_form v φ) → IsFreeFor v t φ → SPA_Deduction_Internal s Γ (substForm v t φ)

| eq_refl (Γ : Set (ArithForm s)) (t : ArithTerm s) : SPA_Deduction_Internal s Γ (ArithForm.eq t t)

| eq_symm (Γ : Set (ArithForm s)) (t1 t2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (ArithForm.eq t2 t1)

| eq_trans (Γ : Set (ArithForm s)) (t1 t2 t3 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (ArithForm.eq t2 t3) → SPA_Deduction_Internal s Γ (ArithForm.eq t1 t3)

| eq_succ (Γ : Set (ArithForm s)) (t1 t2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (ArithForm.eq (ArithTerm.succ t1) (ArithTerm.succ t2))

| eq_add (Γ : Set (ArithForm s)) (t1 t2 u1 u2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (ArithForm.eq u1 u2) → SPA_Deduction_Internal s Γ (ArithForm.eq (ArithTerm.add t1 u1) (ArithTerm.add t2 u2))

| eq_mul (Γ : Set (ArithForm s)) (t1 t2 u1 u2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (ArithForm.eq u1 u2) → SPA_Deduction_Internal s Γ (ArithForm.eq (ArithTerm.mul t1 u1) (ArithTerm.mul t2 u2))

| eq_exp (Γ : Set (ArithForm s)) (t1 t2 u1 u2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (ArithForm.eq u1 u2) → SPA_Deduction_Internal s Γ (ArithForm.eq (ArithTerm.exp t1 u1) (ArithTerm.exp t2 u2))

| eq_pred (Γ : Set (ArithForm s)) (t1 t2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (ArithForm.eq (ArithTerm.pred t1) (ArithTerm.pred t2))

| eq_provable_op (Γ : Set (ArithForm s)) {s_l : Nat} (h_eq : s = s_l + 1) (t1 t2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (ArithForm.provable_op h_eq t1) → SPA_Deduction_Internal s Γ (ArithForm.provable_op h_eq t2)

| eq_true_op (Γ : Set (ArithForm s)) {s_l : Nat} (h_eq : s = s_l + 1) (t1 t2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (ArithForm.true_op h_eq t1) → SPA_Deduction_Internal s Γ (ArithForm.true_op h_eq t2)

| eq_list_nth_eq_1 (Γ : Set (ArithForm s)) (t1 t2 t3 u1 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 u1) → SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 t3) → SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq u1 t2 t3)

| eq_list_nth_eq_2 (Γ : Set (ArithForm s)) (t1 t2 t3 u2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t2 u2) → SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 t3) → SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 u2 t3)

| eq_list_nth_eq_3 (Γ : Set (ArithForm s)) (t1 t2 t3 u3 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t3 u3) → SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 t3) → SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 u3)

| eq_subst (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s) (t1 t2 : ArithTerm s) :

SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2) → SPA_Deduction_Internal s Γ (substForm v t1 φ) → IsFreeFor v t1 φ → IsFreeFor v t2 φ → SPA_Deduction_Internal s Γ (substForm v t2 φ)

| weaken (Γ : Set (ArithForm s)) (φ : ArithForm s) (Γ' : Set (ArithForm s)) :

SPA_Deduction_Internal s Γ φ → Γ ⊆ Γ' → SPA_Deduction_Internal s Γ' φ

| ax_peano1 (Γ : Set (ArithForm s)) (x : ArithTerm s) : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.eq (ArithTerm.succ x) ArithTerm.zero))

| ax_peano2 (Γ : Set (ArithForm s)) (x y : ArithTerm s) : SPA_Deduction_Internal s Γ (ArithForm.impl (ArithForm.eq (ArithTerm.succ x) (ArithTerm.succ y)) (ArithForm.eq x y))

| induction (Γ : Set (ArithForm s)) (φ : ArithForm s) (v : Nat) :

SPA_Deduction_Internal s Γ (substForm v ArithTerm.zero φ) → SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.impl φ (substForm v (ArithTerm.succ (ArithTerm.var v)) φ))) → SPA_Deduction_Internal s Γ (ArithForm.forall_form v φ)

Predicados auxiliares sobre derivaciones: frescura y cota de variables

Se definen dos predicados inductivos sobre el sistema deductivo que permiten razonar sobre propiedades de las derivaciones: uno que garantiza que una variable fresca no aparece en toda una derivación, y otro que acota superiormente las variables que aparecen en una derivación.

Predicado inductivo `FreshForDerivation`: Variable fresca para una derivación

Establece que una variable w no ocurre en ninguna fórmula o término que aparezca en una derivación. Para cada regla de inferencia, se especifican las condiciones necesarias:

$$ \text{FreshForDerivation } w\ d : \text{Prop} $$

donde d es una derivación en `SPA_Deduction_Internal`. Las condiciones clave incluyen:

- **`hyp`:** w no ocurre en la fórmula φ utilizada como hipótesis.

- **`imp_intro`, `neg_intro`:** w no ocurre en la fórmula φ que se descarga.

- **`forall_intro`:** w es distinta de la variable generalizada v.

- **`forall_elim`:** w es distinta de v y no ocurre en el término t que se sustituye.

- **`eq_subst`:** w es distinta de v, no ocurre en φ, y las subderivaciones son frescas.

- **`induction`:** w es distinta de v, no ocurre en φ, y las subderivaciones son frescas.

- **`ax_peano1`, `ax_peano2`:** w no ocurre en los términos involucrados.

Intuición y necesidad

Este predicado es una herramienta técnica para demostrar que ciertas variables pueden considerarse "frescas" respecto a una derivación completa. Es utilizado en los

lemas de cambio de variable ligada y en la construcción de testigos de Henkin, donde se necesita garantizar que la nueva variable elegida no interfiere con ninguna parte de las derivaciones previas.

Predicado inductivo `MaxVarDeriv`: Cota superior de variables en una derivación

Asocia a cada derivación d un número natural m que es una cota superior para todas las variables (libres o ligadas) que aparecen en d:

$$ \text{MaxVarDeriv } d\ m : \text{Prop} $$

Para cada regla de inferencia, se define cómo se combinan las cotas de las subderivaciones:

- **`hyp`:** La cota es `max_var_form(φ)`.

- **`imp_intro`:** El máximo entre `max_var_form(φ)` y la cota de la subderivación.

- **`forall_intro`:** El máximo entre v y la cota de la subderivación (pues v aparece como variable ligada).

- **`forall_elim`:** El máximo entre v, `max_var_term(t)`, y la cota de la subderivación.

- **`induction`:** El máximo entre v, `max_var_form(φ)`, y las cotas de ambas subderivaciones.

- **Reglas de congruencia:** El máximo de las cotas de las subderivaciones (pues no introducen nuevas variables).

Intuición y necesidad

`MaxVarDeriv` proporciona un mecanismo para generar variables frescas respecto a una derivación completa: cualquier variable con índice mayor que la cota m está garantizada como no ocurriente en toda la derivación. Esta propiedad es fundamental para la construcción de la secuencia de Henkin, donde se necesita elegir variables y constantes de Henkin que no colisionen con ninguna fórmula utilizada hasta el

momento. La definición inductiva permite calcular la cota de manera composicional a partir de las cotas de las subderivaciones.

Significado conjunto

`FreshForDerivation` y `MaxVarDeriv` son dos caras de la misma moneda: mientras que `FreshForDerivation` establece directamente que una variable concreta w es fresca para una derivación, `MaxVarDeriv` proporciona una cota numérica que permite generar infinitas variables frescas (todas las mayores que la cota). Ambos predicados son utilizados en los lemas de cambio de variable ligada (`change_var_equivalence_strong`) y en las demostraciones de preservación de la consistencia durante la extensión de Henkin, donde se requiere que los nuevos símbolos no interfieran con las deducciones ya establecidas. La definición detallada para cada regla refleja un análisis cuidadoso de qué variables aparecen en cada construcción deductiva.

inductive FreshForDerivation {s : Nat} (w : Nat) : {Γ : Set (ArithForm s)} → {φ : ArithForm s} → SPA_Deduction_Internal s Γ φ → Prop where

| hyp (Γ : Set (ArithForm s)) (φ : ArithForm s) (h : φ ∈ Γ) (h_w : NotOccurForm w φ) : FreshForDerivation w (SPA_Deduction_Internal.hyp Γ φ h)

| imp_intro (Γ : Set (ArithForm s)) (φ ψ : ArithForm s) (h : SPA_Deduction_Internal s (insert φ Γ) ψ) (h_wφ : NotOccurForm w φ) (ih : FreshForDerivation w h) : FreshForDerivation w (SPA_Deduction_Internal.imp_intro Γ φ ψ h)

| imp_elim (Γ : Set (ArithForm s)) (φ ψ : ArithForm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.impl φ ψ)) (h2 : SPA_Deduction_Internal s Γ φ) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.imp_elim Γ φ ψ h1 h2)

| double_neg (Γ : Set (ArithForm s)) (φ : ArithForm s) (h : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.neg φ))) (ih : FreshForDerivation w h) : FreshForDerivation w (SPA_Deduction_Internal.double_neg Γ φ h)

| neg_intro (Γ : Set (ArithForm s)) (φ : ArithForm s) (h : SPA_Deduction_Internal s (insert φ Γ) ArithForm.bottom) (h_wφ : NotOccurForm w φ) (ih : FreshForDerivation w h) : FreshForDerivation w (SPA_Deduction_Internal.neg_intro Γ φ h)

| neg_elim (Γ : Set (ArithForm s)) (φ : ArithForm s) (h1 : SPA_Deduction_Internal s Γ φ) (h2 : SPA_Deduction_Internal s Γ (ArithForm.neg φ)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.neg_elim Γ φ h1 h2)

| forall_intro (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s) (h : SPA_Deduction_Internal s Γ φ) (h_vΓ : NotFreeInSet v Γ) (hnw : v ≠ w) (ih : FreshForDerivation w h) : FreshForDerivation w (SPA_Deduction_Internal.forall_intro Γ v φ h h_vΓ)

| forall_elim (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s) (t : ArithTerm s) (h : SPA_Deduction_Internal s Γ (ArithForm.forall_form v φ)) (h_free : IsFreeFor v t φ) (hnw : v ≠ w) (h_wt : NotOccur w t) (ih : FreshForDerivation w h) : FreshForDerivation w (SPA_Deduction_Internal.forall_elim Γ v φ t h h_free)

| eq_refl (Γ : Set (ArithForm s)) (t : ArithTerm s) (h_wt : NotOccur w t) : FreshForDerivation w (SPA_Deduction_Internal.eq_refl Γ t)

| eq_symm (Γ : Set (ArithForm s)) (t1 t2 : ArithTerm s) (h : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (ih : FreshForDerivation w h) : FreshForDerivation w (SPA_Deduction_Internal.eq_symm Γ t1 t2 h)

| eq_trans (Γ : Set (ArithForm s)) (t1 t2 t3 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.eq t2 t3)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_trans Γ t1 t2 t3 h1 h2)

| eq_succ (Γ : Set (ArithForm s)) (t1 t2 : ArithTerm s) (h : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (ih : FreshForDerivation w h) : FreshForDerivation w (SPA_Deduction_Internal.eq_succ Γ t1 t2 h)

| eq_add (Γ : Set (ArithForm s)) (t1 t2 u1 u2 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ

(ArithForm.eq u1 u2)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_add Γ t1 t2 u1 u2 h1 h2)

| eq_mul (Γ : Set (ArithForm s)) (t1 t2 u1 u2 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.eq u1 u2)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_mul Γ t1 t2 u1 u2 h1 h2)

| eq_exp (Γ : Set (ArithForm s)) (t1 t2 u1 u2 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.eq u1 u2)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_exp Γ t1 t2 u1 u2 h1 h2)

| eq_pred (Γ : Set (ArithForm s)) (t1 t2 : ArithTerm s) (h : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (ih : FreshForDerivation w h) : FreshForDerivation w (SPA_Deduction_Internal.eq_pred Γ t1 t2 h)

| eq_provable_op (Γ : Set (ArithForm s)) {s_l : Nat} (h_eq : s = s_l + 1) (t1 t2 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.provable_op h_eq t1)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_provable_op Γ h_eq t1 t2 h1 h2)

| eq_true_op (Γ : Set (ArithForm s)) {s_l : Nat} (h_eq : s = s_l + 1) (t1 t2 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.true_op h_eq t1)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_true_op Γ h_eq t1 t2 h1 h2)

| eq_list_nth_eq_1 (Γ : Set (ArithForm s)) (t1 t2 t3 u1 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 u1)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 t3)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_list_nth_eq_1 Γ t1 t2 t3 u1 h1 h2)

| eq_list_nth_eq_2 (Γ : Set (ArithForm s)) (t1 t2 t3 u2 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t2 u2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 t3)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_list_nth_eq_2 Γ t1 t2 t3 u2 h1 h2)

| eq_list_nth_eq_3 (Γ : Set (ArithForm s)) (t1 t2 t3 u3 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t3 u3)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 t3)) (ih1 : FreshForDerivation w h1) (ih2 : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_list_nth_eq_3 Γ t1 t2 t3 u3 h1 h2)

| eq_subst (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s) (t1 t2 : ArithTerm s) (h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ

(substForm v t1 φ)) (h_free1 : IsFreeFor v t1 φ) (h_free2 : IsFreeFor v t2 φ) (hnw : v ≠ w) (h_wφ : NotOccurForm w φ) (ih1_h : FreshForDerivation w h1) (ih2_h : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.eq_subst Γ v φ t1 t2 h1 h2 h_free1 h_free2)

| weaken (Γ : Set (ArithForm s)) (φ : ArithForm s) (Γ' : Set (ArithForm s)) (h : SPA_Deduction_Internal s Γ φ) (h_sub : Γ ⊆ Γ') (ih : FreshForDerivation w h) : FreshForDerivation w (SPA_Deduction_Internal.weaken Γ φ Γ' h h_sub)

| ax_peano1 (Γ : Set (ArithForm s)) (x : ArithTerm s) (h_wx : NotOccur w x) : FreshForDerivation w (SPA_Deduction_Internal.ax_peano1 Γ x)

| ax_peano2 (Γ : Set (ArithForm s)) (x y : ArithTerm s) (h_wx : NotOccur w x) (h_wy : NotOccur w y) : FreshForDerivation w (SPA_Deduction_Internal.ax_peano2 Γ x y)

| induction (Γ : Set (ArithForm s)) (φ : ArithForm s) (v : Nat) (h1 : SPA_Deduction_Internal s Γ (substForm v ArithTerm.zero φ)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.impl φ (substForm v (ArithTerm.succ (ArithTerm.var v)) φ)))) (hnw : v ≠ w) (h_wφ : NotOccurForm w φ) (ih1_h : FreshForDerivation w h1) (ih2_h : FreshForDerivation w h2) : FreshForDerivation w (SPA_Deduction_Internal.induction Γ φ v h1 h2)

inductive MaxVarDeriv {s : Nat} : {Γ : Set (ArithForm s)} → {φ : ArithForm s} → SPA_Deduction_Internal s Γ φ → Nat → Prop where

| hyp (Γ : Set (ArithForm s)) (φ : ArithForm s) (h : φ ∈ Γ) :

MaxVarDeriv (SPA_Deduction_Internal.hyp Γ φ h) (max_var_form φ)

| imp_intro (Γ : Set (ArithForm s)) (φ ψ : ArithForm s) (h : SPA_Deduction_Internal s (insert φ Γ) ψ)

(m : Nat) (h_sub : MaxVarDeriv h m) :

MaxVarDeriv (SPA_Deduction_Internal.imp_intro Γ φ ψ h) (Nat.max (max_var_form φ) m)

| imp_elim (Γ : Set (ArithForm s)) (φ ψ : ArithForm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.impl φ ψ)) (h2 : SPA_Deduction_Internal s Γ φ)

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.imp_elim Γ φ ψ h1 h2) (Nat.max m1 m2)

| double_neg (Γ : Set (ArithForm s)) (φ : ArithForm s)

(h : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.neg φ)))

(m : Nat) (h_sub : MaxVarDeriv h m) :

MaxVarDeriv (SPA_Deduction_Internal.double_neg Γ φ h) m

| neg_intro (Γ : Set (ArithForm s)) (φ : ArithForm s)

(h : SPA_Deduction_Internal s (insert φ Γ) ArithForm.bottom)

(m : Nat) (h_sub : MaxVarDeriv h m) :

MaxVarDeriv (SPA_Deduction_Internal.neg_intro Γ φ h) (Nat.max (max_var_form φ) m)

| neg_elim (Γ : Set (ArithForm s)) (φ : ArithForm s)

(h1 : SPA_Deduction_Internal s Γ φ) (h2 : SPA_Deduction_Internal s Γ (ArithForm.neg φ))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.neg_elim Γ φ h1 h2) (Nat.max m1 m2)

| forall_intro (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s)

(h : SPA_Deduction_Internal s Γ φ) (h_vΓ : NotFreeInSet v Γ)

(m : Nat) (h_sub : MaxVarDeriv h m) :

MaxVarDeriv (SPA_Deduction_Internal.forall_intro Γ v φ h h_vΓ) (Nat.max v m)

| forall_elim (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s) (t : ArithTerm s)

(h : SPA_Deduction_Internal s Γ (ArithForm.forall_form v φ)) (h_free : IsFreeFor v t φ)

(m : Nat) (h_sub : MaxVarDeriv h m) :

MaxVarDeriv (SPA_Deduction_Internal.forall_elim Γ v φ t h h_free) (Nat.max v (Nat.max (max_var_term t) m))

| eq_refl (Γ : Set (ArithForm s)) (t : ArithTerm s) :

MaxVarDeriv (SPA_Deduction_Internal.eq_refl Γ t) (max_var_term t)

| eq_symm (Γ : Set (ArithForm s)) (t1 t2 : ArithTerm s)

(h : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2))

(m : Nat) (h_sub : MaxVarDeriv h m) :

MaxVarDeriv (SPA_Deduction_Internal.eq_symm Γ t1 t2 h) m

| eq_trans (Γ : Set (ArithForm s)) (t1 t2 t3 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.eq t2 t3))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_trans Γ t1 t2 t3 h1 h2) (Nat.max m1 m2)

| eq_succ (Γ : Set (ArithForm s)) (t1 t2 : ArithTerm s)

(h : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2))

(m : Nat) (h_sub : MaxVarDeriv h m) :

MaxVarDeriv (SPA_Deduction_Internal.eq_succ Γ t1 t2 h) m

| eq_add (Γ : Set (ArithForm s)) (t1 t2 u1 u2 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.eq u1 u2))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_add Γ t1 t2 u1 u2 h1 h2) (Nat.max m1 m2)

| eq_mul (Γ : Set (ArithForm s)) (t1 t2 u1 u2 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.eq u1 u2))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_mul Γ t1 t2 u1 u2 h1 h2) (Nat.max m1 m2)

| eq_exp (Γ : Set (ArithForm s)) (t1 t2 u1 u2 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.eq u1 u2))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_exp Γ t1 t2 u1 u2 h1 h2) (Nat.max m1 m2)

| eq_pred (Γ : Set (ArithForm s)) (t1 t2 : ArithTerm s)

(h : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2))

(m : Nat) (h_sub : MaxVarDeriv h m) :

MaxVarDeriv (SPA_Deduction_Internal.eq_pred Γ t1 t2 h) m

| eq_provable_op (Γ : Set (ArithForm s)) {s_l : Nat} (h_eq : s = s_l + 1) (t1 t2 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.provable_op h_eq t1))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_provable_op Γ h_eq t1 t2 h1 h2) (Nat.max m1 m2)

| eq_true_op (Γ : Set (ArithForm s)) {s_l : Nat} (h_eq : s = s_l + 1) (t1 t2 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.true_op h_eq t1))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_true_op Γ h_eq t1 t2 h1 h2) (Nat.max m1 m2)

| eq_list_nth_eq_1 (Γ : Set (ArithForm s)) (t1 t2 t3 u1 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 u1)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 t3))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_list_nth_eq_1 Γ t1 t2 t3 u1 h1 h2) (Nat.max m1 m2)

| eq_list_nth_eq_2 (Γ : Set (ArithForm s)) (t1 t2 t3 u2 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t2 u2)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 t3))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_list_nth_eq_2 Γ t1 t2 t3 u2 h1 h2) (Nat.max m1 m2)

| eq_list_nth_eq_3 (Γ : Set (ArithForm s)) (t1 t2 t3 u3 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t3 u3)) (h2 : SPA_Deduction_Internal s Γ (ArithForm.list_nth_eq t1 t2 t3))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_list_nth_eq_3 Γ t1 t2 t3 u3 h1 h2) (Nat.max m1 m2)

| eq_subst (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s) (t1 t2 : ArithTerm s)

(h1 : SPA_Deduction_Internal s Γ (ArithForm.eq t1 t2)) (h2 : SPA_Deduction_Internal s Γ (substForm v t1 φ))

(h_free1 : IsFreeFor v t1 φ) (h_free2 : IsFreeFor v t2 φ)

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.eq_subst Γ v φ t1 t2 h1 h2 h_free1 h_free2) (Nat.max v (Nat.max (max_var_form φ) (Nat.max m1 m2)))

| weaken (Γ : Set (ArithForm s)) (φ : ArithForm s) (Γ' : Set (ArithForm s))

(h : SPA_Deduction_Internal s Γ φ) (h_sub : Γ ⊆ Γ')

(m : Nat) (h_sub_deriv : MaxVarDeriv h m) :

MaxVarDeriv (SPA_Deduction_Internal.weaken Γ φ Γ' h h_sub) m

| ax_peano1 (Γ : Set (ArithForm s)) (x : ArithTerm s) :

MaxVarDeriv (SPA_Deduction_Internal.ax_peano1 Γ x) (max_var_term x)

| ax_peano2 (Γ : Set (ArithForm s)) (x y : ArithTerm s) :

MaxVarDeriv (SPA_Deduction_Internal.ax_peano2 Γ x y) (Nat.max (max_var_term x) (max_var_term y))

| induction (Γ : Set (ArithForm s)) (φ : ArithForm s) (v : Nat)

(h1 : SPA_Deduction_Internal s Γ (substForm v ArithTerm.zero φ))

(h2 : SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.impl φ (substForm v (ArithTerm.succ (ArithTerm.var v)) φ))))

(m1 m2 : Nat) (h_sub1 : MaxVarDeriv h1 m1) (h_sub2 : MaxVarDeriv h2 m2) :

MaxVarDeriv (SPA_Deduction_Internal.induction Γ φ v h1 h2) (Nat.max v (Nat.max (max_var_form φ) (Nat.max m1 m2)))

Lema `FreshForDerivation_of_gt_max_var`: Generación de variable fresca por cota superior

Se demuestra el teorema fundamental que conecta los dos predicados sobre derivaciones: si una derivación tiene una cota superior m para sus variables (según `MaxVarDeriv`), entonces cualquier variable $w > m$ es fresca para toda la derivación (según `FreshForDerivation`):

$$ \forall d : \text{SPA_Deduction_Internal } s\ \Gamma\ \varphi,\ \forall m \in \mathbb{N},\ \text{MaxVarDeriv } d\ m \implies \forall w > m,\ \text{FreshForDerivation } w\ d $$

Estructura de la demostración

La prueba procede por inducción sobre la derivación de `MaxVarDeriv d m`, cubriendo exhaustivamente todas las reglas de inferencia (más de treinta casos). Para cada regla, se utiliza la hipótesis de inducción sobre las subderivaciones y se verifican las condiciones adicionales de frescura requeridas por el constructor correspondiente de `FreshForDerivation`:

- **Cotas de subderivaciones:** En reglas con múltiples premisas (como `imp_elim`, `eq_trans`, `eq_add`, etc.), la cota m es el máximo de las cotas de las subderivaciones. Dado que $w > m$, se tiene $w > m_1$ y $w > m_2$, por lo que las hipótesis de inducción se aplican a ambas subderivaciones.

- **Cotas de fórmulas y términos:** En reglas que introducen nuevas fórmulas o términos (como `hyp`, `imp_intro`, `forall_elim`, `induction`), la cota m incluye también `max_var_form` o `max_var_term` de los componentes sintácticos

involucrados. Dado que $w > m$, se tiene $w > \text{max_var_form}(\varphi)$ y $w > \text{max_var_term}(t)$, lo que permite aplicar `NotOccurForm_of_gt_max_var` y `NotOccur_above_max_var_term` para obtener las condiciones de no ocurrencia requeridas.

- **Variables ligadas:** En reglas que ligan variables (`forall_intro`, `forall_elim`, `induction`), la cota incluye la variable ligada v. Dado que $w > m \ge v$, se tiene $w \neq v$, satisfaciendo la condición de que la variable fresca es distinta de la variable ligada.

Significado

Este lema es el puente entre la noción semántica de cota (`MaxVarDeriv`) y la noción sintáctica de frescura (`FreshForDerivation`). Proporciona un método constructivo para generar variables frescas respecto a una derivación: basta calcular una cota superior m (lo cual es posible porque `MaxVarDeriv` es un predicado inductivo que puede ser habitado para cualquier derivación concreta) y elegir $w = m + 1$ (o cualquier número mayor).

La demostración, aunque extensa por el número de casos, es mecánica: en cada caso se descomponen las desigualdades usando las propiedades de `Nat.le_max_left` y `Nat.le_max_right` para deducir que w supera cada componente individual, y luego se aplican las hipótesis de inducción y los lemas de no ocurrencia. Esta sistematicidad refleja el cuidado puesto en el diseño de `MaxVarDeriv`, cuyas cotas están calibradas precisamente para que este lema sea demostrable por inducción directa.

La importancia de este resultado para el desarrollo global radica en que garantiza un suministro ilimitado de variables frescas para cualquier construcción deductiva, lo cual es indispensable para los lemas de cambio de variable ligada, la construcción de testigos de Henkin, y las demostraciones de inducción interna que requieren elegir variables que no interfieran con el contexto deductivo existente.

```
lemma FreshForDerivation_of_gt_max_var {s : Nat} {Γ : Set (ArithForm s)} {φ : ArithForm s}
  (h : SPA_Deduction_Internal s Γ φ) (m : Nat) (hmax : MaxVarDeriv h m) (w : Nat) (hw : w > m) :
  FreshForDerivation w h := by
 induction hmax with
 | hyp Γ φ h_in =>
  exact FreshForDerivation.hyp Γ φ h_in (NotOccurForm_of_gt_max_var φ w hw)
 | imp_intro Γ φ ψ h m_sub hmax_sub ih =>
  have h_phi : max_var_form φ ≤ Nat.max (max_var_form φ) m_sub := Nat.le_max_left _ _
  have h_m : m_sub ≤ Nat.max (max_var_form φ) m_sub := Nat.le_max_right _ _
  exact FreshForDerivation.imp_intro Γ φ ψ h
   (NotOccurForm_of_gt_max_var φ w (by omega))
   (ih (by omega))
 | imp_elim Γ φ ψ h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
  have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
  have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
  exact FreshForDerivation.imp_elim Γ φ ψ h1 h2
   (ih1 (by omega))
   (ih2 (by omega))
 | double_neg Γ φ h m_sub hmax_sub ih =>
```

```
    exact FreshForDerivation.double_neg Γ φ h (ih hw)
  | neg_intro Γ φ h m_sub hmax_sub ih =>
    have h_phi : max_var_form φ ≤ Nat.max (max_var_form φ) m_sub := Nat.le_max_left _
_
    have h_m : m_sub ≤ Nat.max (max_var_form φ) m_sub := Nat.le_max_right _ _
    exact FreshForDerivation.neg_intro Γ φ h
      (NotOccurForm_of_gt_max_var φ w (by omega))
      (ih (by omega))
  | neg_elim Γ φ h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
    have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
    have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
    exact FreshForDerivation.neg_elim Γ φ h1 h2
      (ih1 (by omega))
      (ih2 (by omega))
  | forall_intro Γ v φ h h_vΓ m_sub hmax_sub ih =>
    have h_v : v ≤ Nat.max v m_sub := Nat.le_max_left _ _
    have h_m : m_sub ≤ Nat.max v m_sub := Nat.le_max_right _ _
    exact FreshForDerivation.forall_intro Γ v φ h h_vΓ
      (by omega)
      (ih (by omega))
  | forall_elim Γ v φ t h h_free m_sub hmax_sub ih =>
    have h_v : v ≤ Nat.max v (Nat.max (max_var_term t) m_sub) := Nat.le_max_left _ _
    have h_rest : Nat.max (max_var_term t) m_sub ≤ Nat.max v (Nat.max (max_var_term
t) m_sub) := Nat.le_max_right _ _
    have h_t : max_var_term t ≤ Nat.max (max_var_term t) m_sub := Nat.le_max_left _ _
    have h_m : m_sub ≤ Nat.max (max_var_term t) m_sub := Nat.le_max_right _ _
    exact FreshForDerivation.forall_elim Γ v φ t h h_free
      (by omega)
      (NotOccur_above_max_var_term t w (by omega))
```

```
    (ih (by omega))
  | eq_refl Γ t =>
    exact FreshForDerivation.eq_refl Γ t (NotOccur_above_max_var_term t w hw)
  | eq_symm Γ t1 t2 h m_sub hmax_sub ih =>
    exact FreshForDerivation.eq_symm Γ t1 t2 h (ih hw)
  | eq_trans Γ t1 t2 t3 h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
    have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
    have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
    exact FreshForDerivation.eq_trans Γ t1 t2 t3 h1 h2
      (ih1 (by omega))
      (ih2 (by omega))
  | eq_succ Γ t1 t2 h m_sub hmax_sub ih =>
    exact FreshForDerivation.eq_succ Γ t1 t2 h (ih hw)
  | eq_add Γ t1 t2 u1 u2 h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
    have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
    have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
    exact FreshForDerivation.eq_add Γ t1 t2 u1 u2 h1 h2
      (ih1 (by omega))
      (ih2 (by omega))
  | eq_mul Γ t1 t2 u1 u2 h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
    have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
    have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
    exact FreshForDerivation.eq_mul Γ t1 t2 u1 u2 h1 h2
      (ih1 (by omega))
      (ih2 (by omega))
  | eq_exp Γ t1 t2 u1 u2 h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
    have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
    have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
```

```
  exact FreshForDerivation.eq_exp Γ t1 t2 u1 u2 h1 h2
    (ih1 (by omega))
    (ih2 (by omega))
| eq_pred Γ t1 t2 h m_sub hmax_sub ih =>
  exact FreshForDerivation.eq_pred Γ t1 t2 h (ih hw)
| eq_provable_op Γ h_eq t1 t2 h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
  have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
  have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
  exact FreshForDerivation.eq_provable_op Γ h_eq t1 t2 h1 h2
    (ih1 (by omega))
    (ih2 (by omega))
| eq_true_op Γ h_eq t1 t2 h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
  have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
  have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
  exact FreshForDerivation.eq_true_op Γ h_eq t1 t2 h1 h2
    (ih1 (by omega))
    (ih2 (by omega))
| eq_list_nth_eq_1 Γ t1 t2 t3 u1 h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
  have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
  have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
  exact FreshForDerivation.eq_list_nth_eq_1 Γ t1 t2 t3 u1 h1 h2
    (ih1 (by omega))
    (ih2 (by omega))
| eq_list_nth_eq_2 Γ t1 t2 t3 u2 h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
  have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
  have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
  exact FreshForDerivation.eq_list_nth_eq_2 Γ t1 t2 t3 u2 h1 h2
    (ih1 (by omega))
```

```
    (ih2 (by omega))
  | eq_list_nth_eq_3 Γ t1 t2 t3 u3 h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
    have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
    have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
    exact FreshForDerivation.eq_list_nth_eq_3 Γ t1 t2 t3 u3 h1 h2
      (ih1 (by omega))
      (ih2 (by omega))
  | eq_subst Γ v φ t1 t2 h1 h2 h_free1 h_free2 m1 m2 hmax1 hmax2 ih1 ih2 =>
    have h_v : v ≤ Nat.max v (Nat.max (max_var_form φ) (Nat.max m1 m2)) :=
Nat.le_max_left _ _
    have h_rest : Nat.max (max_var_form φ) (Nat.max m1 m2) ≤ Nat.max v (Nat.max
(max_var_form φ) (Nat.max m1 m2)) := Nat.le_max_right _ _
    have h_phi : max_var_form φ ≤ Nat.max (max_var_form φ) (Nat.max m1 m2) :=
Nat.le_max_left _ _
    have h_m12 : Nat.max m1 m2 ≤ Nat.max (max_var_form φ) (Nat.max m1 m2) :=
Nat.le_max_right _ _
    have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
    have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
    exact FreshForDerivation.eq_subst Γ v φ t1 t2 h1 h2 h_free1 h_free2
      (by omega)
      (NotOccurForm_of_gt_max_var φ w (by omega))
      (ih1 (by omega))
      (ih2 (by omega))
  | weaken Γ φ Γ' h h_sub m_sub hmax_sub ih =>
    exact FreshForDerivation.weaken Γ φ Γ' h h_sub (ih hw)
  | ax_peano1 Γ x =>
    exact FreshForDerivation.ax_peano1 Γ x (NotOccur_above_max_var_term x w hw)
  | ax_peano2 Γ x y =>
    have h_x : max_var_term x ≤ Nat.max (max_var_term x) (max_var_term y) :=
Nat.le_max_left _ _
```

```
  have h_y : max_var_term y ≤ Nat.max (max_var_term x) (max_var_term y) := Nat.le_max_right _ _
  exact FreshForDerivation.ax_peano2 Γ x y
    (NotOccur_above_max_var_term x w (by omega))
    (NotOccur_above_max_var_term y w (by omega))
| induction Γ φ v h1 h2 m1 m2 hmax1 hmax2 ih1 ih2 =>
  have h_v : v ≤ Nat.max v (Nat.max (max_var_form φ) (Nat.max m1 m2)) := Nat.le_max_left _ _
  have h_rest : Nat.max (max_var_form φ) (Nat.max m1 m2) ≤ Nat.max v (Nat.max (max_var_form φ) (Nat.max m1 m2)) := Nat.le_max_right _ _
  have h_phi : max_var_form φ ≤ Nat.max (max_var_form φ) (Nat.max m1 m2) := Nat.le_max_left _ _
  have h_m12 : Nat.max m1 m2 ≤ Nat.max (max_var_form φ) (Nat.max m1 m2) := Nat.le_max_right _ _
  have h_m1 : m1 ≤ Nat.max m1 m2 := Nat.le_max_left _ _
  have h_m2 : m2 ≤ Nat.max m1 m2 := Nat.le_max_right _ _
  exact FreshForDerivation.induction Γ φ v h1 h2
    (by omega)
    (NotOccurForm_of_gt_max_var φ w (by omega))
    (ih1 (by omega))
    (ih2 (by omega))
```

Lema `exists_max_var_deriv`: Existencia de cota superior para cualquier derivación

Se demuestra que toda derivación en el sistema `SPA_Deduction_Internal` tiene asociada una cota superior para las variables que aparecen en ella, según el predicado `MaxVarDeriv`:

$$ \forall d : \text{SPA_Deduction_Internal } s\ \Gamma\ \varphi,\ \exists m \in \mathbb{N},\ \text{MaxVarDeriv } d\ m $$

Estructura de la demostración

La prueba procede por inducción sobre la derivación d, cubriendo cada regla de inferencia del sistema deductivo. Para cada caso, se construye una cota m a partir de las cotas de las subderivaciones (obtenidas por hipótesis de inducción) y de los máximos de variables de las fórmulas y términos involucrados:

- **Reglas sin premisas (`hyp`, `eq_refl`, `ax_peano1`, `ax_peano2`):** La cota es simplemente `max_var_form(φ)` o `max_var_term(t)` (o el máximo de ambos para `ax_peano2`).

- **Reglas unarias (`double_neg`, `neg`, `eq_symm`, `eq_succ`, `eq_pred`, `weaken`):** La cota se hereda directamente de la subderivación, pues estas reglas no introducen nuevas variables.

- **Reglas binarias (`imp_elim`, `neg_elim`, `eq_trans`, `eq_add`, `eq_mul`, `eq_exp`, `eq_provable_op`, `eq_true_op`, `eq_list_nth_eq_*`):** La cota es el máximo de las cotas de ambas subderivaciones.

- **Reglas que introducen variables nuevas:**

- `imp_intro`, `neg_intro`: El máximo entre `max_var_form(φ)` (la fórmula descargada) y la cota de la subderivación.

- `forall_intro`: El máximo entre la variable ligada v y la cota de la subderivación.

- `forall_elim`: El máximo entre v, `max_var_term(t)` (el término sustituido), y la cota de la subderivación.

- `eq_subst`: El máximo entre v, `max_var_form(φ)`, y las cotas de ambas subderivaciones.

- `induction`: El máximo entre v, `max_var_form(φ)`, y las cotas de ambas subderivaciones.

Significado

`exists_max_var_deriv` es un resultado de carácter metacompacto: establece que el conjunto de variables que aparecen en una derivación es siempre finito y acotado. Esta propiedad es fundamental para la construcción de Henkin, ya que garantiza que siempre es posible elegir una variable o constante de Henkin fresca (con índice mayor que la cota) que no interfiera con ninguna derivación previa.

Combinado con `FreshForDerivation_of_gt_max_var`, este lema proporciona un método efectivo para generar variables frescas: dada cualquier derivación d, se calcula su cota m mediante `exists_max_var_deriv`, y cualquier variable $w > m$ satisface `FreshForDerivation w d`, es decir, es completamente nueva respecto a toda la derivación. Este mecanismo de "frescura por cota superior" es ubicuo en las demostraciones de cambio de variable ligada y en la preservación de la consistencia durante las extensiones de la teoría.

La demostración, aunque mecánica, refleja un análisis composicional preciso de qué variables aparecen en cada regla de inferencia, y constituye un ejemplo de cómo la inducción estructural sobre derivaciones permite establecer propiedades globales a partir de las propiedades locales de cada regla.

```
lemma exists_max_var_deriv {s : Nat} {Γ : Set (ArithForm s)} {φ : ArithForm s}
    (h : SPA_Deduction_Internal s Γ φ) : ∃ m, MaxVarDeriv h m := by
  induction h with
  | hyp Γ φ h_in =>
    use max_var_form φ
    exact MaxVarDeriv.hyp Γ φ h_in
  | imp_intro Γ φ ψ h ih =>
    rcases ih with ⟨m_sub, hmax_sub⟩
    use Nat.max (max_var_form φ) m_sub
    exact MaxVarDeriv.imp_intro Γ φ ψ h m_sub hmax_sub
  | imp_elim Γ φ ψ h1 h2 ih1 ih2 =>
    rcases ih1 with ⟨m1, hmax1⟩
    rcases ih2 with ⟨m2, hmax2⟩
    use Nat.max m1 m2
    exact MaxVarDeriv.imp_elim Γ φ ψ h1 h2 m1 m2 hmax1 hmax2
  | double_neg Γ φ h ih =>
    rcases ih with ⟨m_sub, hmax_sub⟩
    use m_sub
    exact MaxVarDeriv.double_neg Γ φ h m_sub hmax_sub
  | neg_intro Γ φ h ih =>
    rcases ih with ⟨m_sub, hmax_sub⟩
```

```
  use Nat.max (max_var_form φ) m_sub
  exact MaxVarDeriv.neg_intro Γ φ h m_sub hmax_sub
| neg_elim Γ φ h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.neg_elim Γ φ h1 h2 m1 m2 hmax1 hmax2
| forall_intro Γ v φ h h_vΓ ih =>
  rcases ih with ⟨m_sub, hmax_sub⟩
  use Nat.max v m_sub
  exact MaxVarDeriv.forall_intro Γ v φ h h_vΓ m_sub hmax_sub
| forall_elim Γ v φ t h h_free ih =>
  rcases ih with ⟨m_sub, hmax_sub⟩
  use Nat.max v (Nat.max (max_var_term t) m_sub)
  exact MaxVarDeriv.forall_elim Γ v φ t h h_free m_sub hmax_sub
| eq_refl Γ t =>
  use max_var_term t
  exact MaxVarDeriv.eq_refl Γ t
| eq_symm Γ t1 t2 h ih =>
  rcases ih with ⟨m_sub, hmax_sub⟩
  use m_sub
  exact MaxVarDeriv.eq_symm Γ t1 t2 h m_sub hmax_sub
| eq_trans Γ t1 t2 t3 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.eq_trans Γ t1 t2 t3 h1 h2 m1 m2 hmax1 hmax2
| eq_succ Γ t1 t2 h ih =>
```

```
  rcases ih with ⟨m_sub, hmax_sub⟩
  use m_sub
  exact MaxVarDeriv.eq_succ Γ t1 t2 h m_sub hmax_sub
| eq_add Γ t1 t2 u1 u2 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.eq_add Γ t1 t2 u1 u2 h1 h2 m1 m2 hmax1 hmax2
| eq_mul Γ t1 t2 u1 u2 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.eq_mul Γ t1 t2 u1 u2 h1 h2 m1 m2 hmax1 hmax2
| eq_exp Γ t1 t2 u1 u2 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.eq_exp Γ t1 t2 u1 u2 h1 h2 m1 m2 hmax1 hmax2
| eq_pred Γ t1 t2 h ih =>
  rcases ih with ⟨m_sub, hmax_sub⟩
  use m_sub
  exact MaxVarDeriv.eq_pred Γ t1 t2 h m_sub hmax_sub
| eq_provable_op Γ h_eq t1 t2 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.eq_provable_op Γ h_eq t1 t2 h1 h2 m1 m2 hmax1 hmax2
| eq_true_op Γ h_eq t1 t2 h1 h2 ih1 ih2 =>
```

```
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.eq_true_op Γ h_eq t1 t2 h1 h2 m1 m2 hmax1 hmax2
| eq_list_nth_eq_1 Γ t1 t2 t3 u1 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.eq_list_nth_eq_1 Γ t1 t2 t3 u1 h1 h2 m1 m2 hmax1 hmax2
| eq_list_nth_eq_2 Γ t1 t2 t3 u2 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.eq_list_nth_eq_2 Γ t1 t2 t3 u2 h1 h2 m1 m2 hmax1 hmax2
| eq_list_nth_eq_3 Γ t1 t2 t3 u3 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max m1 m2
  exact MaxVarDeriv.eq_list_nth_eq_3 Γ t1 t2 t3 u3 h1 h2 m1 m2 hmax1 hmax2
| eq_subst Γ v φ t1 t2 h1 h2 h_free1 h_free2 ih1 ih2 =>
  rcases ih1 with ⟨m1, hmax1⟩
  rcases ih2 with ⟨m2, hmax2⟩
  use Nat.max v (Nat.max (max_var_form φ) (Nat.max m1 m2))
  exact MaxVarDeriv.eq_subst Γ v φ t1 t2 h1 h2 h_free1 h_free2 m1 m2 hmax1 hmax2
| weaken Γ φ Γ' h h_sub ih =>
  rcases ih with ⟨m_sub, hmax_sub⟩
  use m_sub
  exact MaxVarDeriv.weaken Γ φ Γ' h h_sub m_sub hmax_sub
```

```
  | ax_peano1 Γ x =>
    use max_var_term x
    exact MaxVarDeriv.ax_peano1 Γ x
  | ax_peano2 Γ x y =>
    use Nat.max (max_var_term x) (max_var_term y)
    exact MaxVarDeriv.ax_peano2 Γ x y
  | induction Γ φ v h1 h2 ih1 ih2 =>
    rcases ih1 with ⟨m1, hmax1⟩
    rcases ih2 with ⟨m2, hmax2⟩
    use Nat.max v (Nat.max (max_var_form φ) (Nat.max m1 m2))
    exact MaxVarDeriv.induction Γ φ v h1 h2 m1 m2 hmax1 hmax2
```

Lemas de sustitución y frescura

Se demuestran tres lemas fundamentales que caracterizan el comportamiento de la no ocurrencia bajo sustituciones. Estos resultados establecen que la propiedad de que una variable no ocurre en una expresión se preserva bajo sustituciones, siempre que la variable no ocurra en el término de sustitución.

Lema `NotOccur_subst_self`: No ocurrencia tras sustituir la misma variable

Establece que si una variable x no ocurre en un término u, entonces x tampoco ocurre en el resultado de sustituir x por u en cualquier término t:

$$ \forall t, u \in \text{ArithTerm } s,\ \forall x \in \mathbb{N},\ \text{NotOccur } x\ u \implies \text{NotOccur } x\ (\text{substTerm } x\ u\ t) $$

Intuición

Este lema refleja el hecho de que al sustituir la variable x por u, todas las ocurrencias de x en t son reemplazadas por u. Si x no ocurría en u, entonces ninguna ocurrencia de x "sobrevive" a la sustitución: o bien x es reemplazada por u (que no contiene a x), o bien x no aparecía en t en primer lugar. La demostración procede por inducción estructural sobre t, analizando el caso de `var n` según si $n = x$ o no.

Lema `NotOccur_subst_other`: No ocurrencia bajo sustitución de otra variable

Generaliza el lema anterior al caso en que la variable sustituida v puede ser distinta de la variable x cuya no ocurrencia se quiere preservar:

$$ \forall t, u \in \text{ArithTerm } s,\ \forall v, x \in \mathbb{N},\ \text{NotOccur } x\ t \land \text{NotOccur } x\ u \implies \text{NotOccur } x\ (\text{substTerm } v\ u\ t) $$

Intuición

Si x no ocurre en t ni en u, entonces la sustitución de v por u no puede introducir x, independientemente de si $v = x$ o no. La demostración es por inducción sobre t con inversión de la hipótesis `NotOccur x t` para obtener las condiciones sobre los subtérminos. En el caso `var n`, si $v = n$ se reemplaza por u

(que no contiene a x); si $v \neq n$, la variable se preserva (y no es x por la hipótesis).

Lema `NotOccurForm_subst_other`: No ocurrencia en fórmulas bajo sustitución

Extiende el lema anterior al nivel de fórmulas: si una variable x no ocurre en una fórmula φ ni en un término u, entonces x no ocurre en $\varphi[v := u]$:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, x \in \mathbb{N},\ \forall u \in \text{ArithTerm } s, $$

$$ \text{NotOccurForm } x\ \varphi \land \text{NotOccur } x\ u \implies \text{NotOccurForm } x\ (\text{substForm } v\ u\ \varphi) $$

Estructura de la demostración

La prueba procede por inducción sobre φ, utilizando `NotOccur_subst_other` para los casos que involucran términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`) y las hipótesis de inducción para los casos recursivos (`impl`, `neg`). El caso `forall_form n f` requiere distinguir si $v = n$ (la sustitución no afecta) o $v \neq n$ (se propaga recursivamente preservando la condición $x \neq n$ de `NotOccurForm`).

Significado conjunto

Estos tres lemas forman la base de la maquinaria de "frescura bajo sustituciones" que es indispensable para todo el desarrollo posterior. Garantizan que las operaciones de sustitución son "inocuas" respecto a las variables frescas: si una variable no aparece en los ingredientes de una sustitución, no aparecerá en el resultado. Esta propiedad es utilizada implícitamente en prácticamente todas las demostraciones de representabilidad y manipulación sintáctica, donde se necesita garantizar que las

sustituciones de variables ligadas por términos cerrados (o por variables frescas) no introducen variables no deseadas.

En particular, estos lemas justifican el uso extensivo de `substTerm_fresh` y `substForm_fresh_of_NotFreeForm` en las demostraciones previas: cuando un término es cerrado (`isClosed = true`), ninguna variable ocurre en él, por lo que las sustituciones que lo involucran preservan automáticamente cualquier condición de no ocurrencia. La combinación de estos lemas con `NotOccur_of_isClosed` proporciona un mecanismo robusto para simplificar expresiones con sustituciones, como se ha visto profusamente en el desarrollo.

```
--
==========================================================================
-- LEMAS DE SUSTITUCIÓN Y FRESCURA
--
==========================================================================

lemma NotOccur_subst_self {s : Nat} (t : ArithTerm s) (x : Nat) (u : ArithTerm s)
    (h2 : NotOccur x u) : NotOccur x (substTerm x u t) := by
  induction t with
  | var n =>
    unfold substTerm
    split_ifs with h_v
    · exact h2
    · exact NotOccur.var n h_v
  | const c => exact NotOccur.const _
  | henkin c => exact NotOccur.henkin _
  | zero => exact NotOccur.zero
```

```
  | succ t' ih => unfold substTerm; exact NotOccur.succ _ ih
  | add t1 t2 ih1 ih2 => unfold substTerm; exact NotOccur.add _ _ ih1 ih2
  | mul t1 t2 ih1 ih2 => unfold substTerm; exact NotOccur.mul _ _ ih1 ih2
  | exp t1 t2 ih1 ih2 => unfold substTerm; exact NotOccur.exp _ _ ih1 ih2
  | pred t' ih => unfold substTerm; exact NotOccur.pred _ ih

lemma NotOccur_subst_other {s : Nat} (t : ArithTerm s) (v : Nat) (x : Nat) (u : ArithTerm s)
    (h1 : NotOccur x t) (h2 : NotOccur x u) : NotOccur x (substTerm v u t) := by
  induction t with
  | var n =>
    unfold substTerm
    split_ifs with _h_v
    · exact h2
    · exact h1
  | const c => exact NotOccur.const _
  | henkin c => exact NotOccur.henkin _
  | zero => exact NotOccur.zero
  | succ t' ih =>
    unfold substTerm
    cases h1 with | succ _ h_sub =>
    exact NotOccur.succ _ (ih h_sub)
  | add t1 t2 ih1 ih2 =>
    unfold substTerm
    cases h1 with | add _ _ h_t1 h_t2 =>
    exact NotOccur.add _ _ (ih1 h_t1) (ih2 h_t2)
  | mul t1 t2 ih1 ih2 =>
    unfold substTerm
```

```
  cases h1 with | mul _ _ h_t1 h_t2 =>
  exact NotOccur.mul _ _ (ih1 h_t1) (ih2 h_t2)
 | exp t1 t2 ih1 ih2 =>
  unfold substTerm
  cases h1 with | exp _ _ h_t1 h_t2 =>
  exact NotOccur.exp _ _ (ih1 h_t1) (ih2 h_t2)
 | pred t' ih =>
  unfold substTerm
  cases h1 with | pred _ h_sub =>
  exact NotOccur.pred _ (ih h_sub)


lemma NotOccurForm_subst_other {s : Nat} (φ : ArithForm s) (v : Nat) (x : Nat) (u :
ArithTerm s)
  (h1 : NotOccurForm x φ) (h2 : NotOccur x u) : NotOccurForm x (substForm v u φ) :=
by
 induction φ with
 | bottom => unfold substForm; unfold NotOccurForm; trivial
 | eq t1 t2 =>
  unfold NotOccurForm at h1 ⊢
  unfold substForm
  exact ⟨NotOccur_subst_other t1 v x u h1.1 h2, NotOccur_subst_other t2 v x u h1.2 h2⟩
 | impl f1 f2 ih1 ih2 =>
  unfold NotOccurForm at h1 ⊢
  unfold substForm
  exact ⟨ih1 h1.1, ih2 h1.2⟩
 | neg f ih =>
  unfold NotOccurForm at h1 ⊢
  unfold substForm
  exact ih h1
```

```
  | forall_form n f ih =>
    unfold NotOccurForm at h1 ⊢
    unfold substForm
    split_ifs with _h_v
    · exact h1
    · exact ⟨h1.1, ih h1.2⟩
  | provable_op heq t =>
    unfold NotOccurForm at h1 ⊢
    unfold substForm
    exact NotOccur_subst_other t v x u h1 h2
  | true_op heq t =>
    unfold NotOccurForm at h1 ⊢
    unfold substForm
    exact NotOccur_subst_other t v x u h1 h2
  | list_nth_eq t1 t2 t3 =>
    unfold NotOccurForm at h1 ⊢
    unfold substForm
    exact ⟨NotOccur_subst_other t1 v x u h1.1 h2, NotOccur_subst_other t2 v x u h1.2.1
h2, NotOccur_subst_other t3 v x u h1.2.2 h2⟩
```

Lemas `substTerm_fresh` y `substForm_fresh`: Sustitución inocua por no ocurrencia

Se demuestran dos lemas fundamentales que establecen que si una variable no ocurre en un término o fórmula, entonces la sustitución de esa variable por cualquier término no tiene efecto sobre la expresión.

Lema `substTerm_fresh`: Invarianza de términos bajo sustitución fresca

Establece que si una variable v no ocurre en un término t, entonces sustituir v por cualquier término u deja t inalterado:

$$ \forall t, u \in \text{ArithTerm } s,\ \forall v \in \mathbb{N},\ \text{NotOccur } v\ t \implies \text{substTerm } v\ u\ t = t $$

Estructura de la demostración

La prueba procede por inducción estructural sobre t, analizando cada constructor y utilizando inversión sobre la hipótesis `NotOccur v t` para obtener las condiciones sobre los subtérminos:

- **`var n`:** Si $v = n$, entonces `NotOccur v (var v)` sería falso (porque `var` con el mismo índice es una ocurrencia). Por tanto, $v \neq n$ y la sustitución no modifica la variable.

- **Casos base (`const`, `henkin`, `zero`):** La sustitución no tiene efecto por definición.

- **Casos recursivos (`succ`, `add`, `mul`, `exp`, `pred`):** Se invierte la hipótesis `NotOccur` para obtener que v no ocurre en los subtérminos, se aplican las hipótesis de inducción, y se reconstruye el término.

Intuición

Este lema captura la idea fundamental de que la sustitución solo afecta a las ocurrencias libres de la variable sustituida. Si la variable no aparece en absoluto, la operación es la identidad.

Lema `substForm_fresh`: Invarianza de fórmulas bajo sustitución fresca

Extiende el resultado anterior al nivel de fórmulas: si una variable v no ocurre en una fórmula φ, entonces sustituir v por u deja φ inalterada:

$$ \forall \varphi \in \text{ArithForm } } s,\ \forall v \in \mathbb{N},\ \forall u \in \text{ArithTerm } } s,\ \text{NotOccurForm } } v\ \varphi \implies \text{substForm } } v\ u\ \varphi = \varphi $$

Estructura de la demostración

La prueba procede por inducción sobre φ:

- **`bottom`:** Trivial.

- **`eq t1 t2`, `list_nth_eq`:** Se descompone `NotOccurForm` en las condiciones sobre los subtérminos y se aplica `substTerm_fresh` a cada uno.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción a las subfórmulas.

- **`forall_form n f`:** Si $v = n$, la sustitución no penetra el cuantificador (por definición de `substForm`). Si $v \neq n$, se tiene `NotOccurForm v f` (por la condición `NotOccurForm` para `forall_form`) y se aplica la hipótesis de inducción.

- **`provable_op`, `true_op`:** Se aplica `substTerm_fresh` al término argumento.

Significado conjunto

`substTerm_fresh` y `substForm_fresh` son posiblemente los lemas más ubicuamente utilizados en todo el desarrollo del sistema SPA. Justifican todas las simplificaciones de la forma "la sustitución de v por u en X es X mismo porque v no ocurre en X". Esta propiedad es invocada decenas de veces en las demostraciones de representabilidad, en las simplificaciones de sustituciones encadenadas, y en la verificación de condiciones de frescura para las reglas de inferencia.

En particular, cuando se trabaja con términos cerrados (`isClosed = true`), se tiene `NotOccur v t` para cualquier variable v, por lo que estos lemas garantizan que las sustituciones sobre términos cerrados son siempre la identidad. Esto explica por qué en las demostraciones previas aparecen constantemente pasos de reescritura como `substTerm_fresh (natToTerm A) 1 (natToTerm B) ...`: los numerales son términos cerrados, por lo que ninguna variable ocurre en ellos, y cualquier sustitución es inocua.

La combinación de estos lemas con `NotOccur_of_isClosed` y `NotOccurForm_subst_other` forma el núcleo de la maquinaria de manipulación sintáctica del sistema, permitiendo simplificar drásticamente expresiones complejas que involucran múltiples sustituciones anidadas, como las que aparecen en las definiciones de `NumForm`, `BetaForm`, `PairForm` y sus variantes.

```
lemma substTerm_fresh {s : Nat} (t : ArithTerm s) (v : Nat) (u : ArithTerm s) (h : NotOccur v t) :
  substTerm v u t = t := by
 induction t with
 | var n =>
  unfold substTerm
  split_ifs with h_v
  · subst h_v
    cases h with | var _ h_ne => contradiction
  · rfl
 | const c => rfl
 | henkin c => rfl
 | zero => rfl
 | succ t' ih =>
  unfold substTerm
  cases h with | succ _ h_sub =>
  rw [ih h_sub]
 | add t1 t2 ih1 ih2 =>
  unfold substTerm
  cases h with | add _ _ h1 h2 =>
  rw [ih1 h1, ih2 h2]
 | mul t1 t2 ih1 ih2 =>
  unfold substTerm
  cases h with | mul _ _ h1 h2 =>
  rw [ih1 h1, ih2 h2]
 | exp t1 t2 ih1 ih2 =>
  unfold substTerm
  cases h with | exp _ _ h1 h2 =>
```

```
  rw [ih1 h1, ih2 h2]
| pred t' ih =>
  unfold substTerm
  cases h with | pred _ h_sub =>
  rw [ih h_sub]

lemma substForm_fresh {s : Nat} (φ : ArithForm s) (v : Nat) (u : ArithTerm s) (h : NotOccurForm v φ) :
  substForm v u φ = φ := by
 induction φ with
 | bottom => rfl
 | eq t1 t2 =>
  unfold NotOccurForm at h
  dsimp [substForm]
  rw [substTerm_fresh t1 v u h.1, substTerm_fresh t2 v u h.2]
 | impl f1 f2 ih1 ih2 =>
  unfold NotOccurForm at h
  dsimp [substForm]
  rw [ih1 h.1, ih2 h.2]
 | neg f ih =>
  unfold NotOccurForm at h
  dsimp [substForm]
  rw [ih h]
 | forall_form n f ih =>
  unfold NotOccurForm at h
  dsimp [substForm]
  split_ifs with _h_v
  · rfl
```

```
    · rw [ih h.2]
  | provable_op heq t =>
    unfold NotOccurForm at h
    dsimp [substForm]
    rw [substTerm_fresh t v u h]
  | true_op heq t =>
    unfold NotOccurForm at h
    dsimp [substForm]
    rw [substTerm_fresh t v u h]
  | list_nth_eq t1 t2 t3 =>
    unfold NotOccurForm at h
    dsimp [substForm]
    rw [substTerm_fresh t1 v u h.1, substTerm_fresh t2 v u h.2.1, substTerm_fresh t3 v u
h.2.2]
```

Lemas de reducción definicional para sustituciones y libertad

Se presenta una colección de lemas que explicitan el comportamiento de las funciones `substForm`, `substTerm` e `IsFreeFor` sobre cada constructor sintáctico. Estos lemas

son esencialmente las reglas de cómputo que permiten simplificar expresiones durante las demostraciones.

Lemas para `substForm` y `substTerm`

Cada lema establece la igualdad definicional de la sustitución aplicada a un constructor específico:

- **`substForm_bottom`:** La sustitución en `bottom` es `bottom` (no hay variables que sustituir).

- **`substForm_eq`:** La sustitución en una igualdad se propaga a ambos términos.

- **`substForm_impl`:** La sustitución en una implicación se propaga a ambas subfórmulas.

- **`substForm_neg`:** La sustitución en una negación se propaga a la subfórmula.

- **`substForm_forall_form`:** Si la variable de sustitución coincide con la variable ligada, no se modifica la fórmula; en caso contrario, se sustituye en el cuerpo.

- **`substForm_provable_op`, `substForm_true_op`:** La sustitución se propaga al término argumento, preservando el testigo de estratificación.

- **`substForm_list_nth_eq`:** La sustitución se propaga a los tres términos.

- **`substTerm_var`:** La sustitución en una variable la reemplaza por el término si los índices coinciden; en caso contrario, la preserva.

Lemas para `IsFreeFor`

Cada lema establece la condición de libertad para un constructor específico:

- **`IsFreeFor_bottom`, `IsFreeFor_eq`, `IsFreeFor_provable_op`, `IsFreeFor_true_op`, `IsFreeFor_list_nth_eq`:** Siempre verdadero, pues estos constructores no contienen cuantificadores.

- **`IsFreeFor_impl`:** La conjunción de las condiciones para ambas subfórmulas.

- **`IsFreeFor_neg`:** Se reduce a la condición para la subfórmula.

- **`IsFreeFor_forall_form`:** Si $v = n$, la sustitución no penetra y es segura; si $v \neq n$, se requiere que n no ocurra en t y que t sea libre para v en f.

Significado

Estos lemas, aunque triviales desde el punto de vista demostrativo (todos son `rfl`), son fundamentales para la reescritura sistemática de expresiones. En las demostraciones extensas (como las de `add_comm_Δ`, `succ_mul_Δ`, `num_representable_graph`, etc.), aparecen constantemente pasos de reescritura como `rw [substForm_eq]` o `simp [substTerm_var]` que utilizan estos lemas para desplegar las definiciones y exponer la estructura interna de las fórmulas tras las sustituciones. Sin ellos, las expresiones permanecerían opacas bajo las definiciones recursivas, haciendo imposible el razonamiento ecuacional.

-- LEMAS DE REDUCCIÓN DEFINICIONAL

lemma substForm_bottom {s : Nat} (v : Nat) (t : ArithTerm s) :

  substForm v t ArithForm.bottom = ArithForm.bottom := rfl

```
lemma substForm_eq {s : Nat} (v : Nat) (t : ArithTerm s) (t1 t2 : ArithTerm s) :
  substForm v t (ArithForm.eq t1 t2) = ArithForm.eq (substTerm v t t1) (substTerm v t t2) := rfl

lemma substForm_impl {s : Nat} (v : Nat) (t : ArithTerm s) (f1 f2 : ArithForm s) :
  substForm v t (ArithForm.impl f1 f2) = ArithForm.impl (substForm v t f1) (substForm v t f2) := rfl

lemma substForm_neg {s : Nat} (v : Nat) (t : ArithTerm s) (f : ArithForm s) :
  substForm v t (ArithForm.neg f) = ArithForm.neg (substForm v t f) := rfl

lemma substForm_provable_op {s : Nat} (v : Nat) (t : ArithTerm s) {s_lower : Nat} (h : s = s_lower + 1) (t' : ArithTerm s) :
  substForm v t (ArithForm.provable_op h t') = ArithForm.provable_op h (substTerm v t t') := rfl

lemma substForm_true_op {s : Nat} (v : Nat) (t : ArithTerm s) {s_lower : Nat} (h : s = s_lower + 1) (t' : ArithTerm s) :
  substForm v t (ArithForm.true_op h t') = ArithForm.true_op h (substTerm v t t') := rfl

lemma substForm_list_nth_eq {s : Nat} (v : Nat) (t : ArithTerm s) (t1 t2 t3 : ArithTerm s) :
  substForm v t (ArithForm.list_nth_eq t1 t2 t3) = ArithForm.list_nth_eq (substTerm v t t1) (substTerm v t t2) (substTerm v t t3) := rfl

lemma IsFreeFor_bottom {s : Nat} (v : Nat) (t : ArithTerm s) :
  IsFreeFor v t ArithForm.bottom = True := rfl

lemma IsFreeFor_eq {s : Nat} (v : Nat) (t : ArithTerm s) (t1 t2 : ArithTerm s) :
  IsFreeFor v t (ArithForm.eq t1 t2) = True := rfl
```

```
lemma IsFreeFor_impl {s : Nat} (v : Nat) (t : ArithTerm s) (f1 f2 : ArithForm s) :
  IsFreeFor v t (ArithForm.impl f1 f2) = (IsFreeFor v t f1 ∧ IsFreeFor v t f2) := rfl

lemma IsFreeFor_neg {s : Nat} (v : Nat) (t : ArithTerm s) (f : ArithForm s) :
  IsFreeFor v t (ArithForm.neg f) = IsFreeFor v t f := rfl

lemma IsFreeFor_list_nth_eq {s : Nat} (v : Nat) (t : ArithTerm s) (t1 t2 t3 : ArithTerm s)
:
  IsFreeFor v t (ArithForm.list_nth_eq t1 t2 t3) = True := rfl

lemma IsFreeFor_forall_form {s : Nat} (v : Nat) (t : ArithTerm s) (n : Nat) (f : ArithForm
s) :
  IsFreeFor v t (ArithForm.forall_form n f) = (if v = n then True else NotOccur n t ∧
IsFreeFor v t f) := rfl

lemma substForm_forall_form {s : Nat} (v : Nat) (u : ArithTerm s) (n : Nat) (f :
ArithForm s) :
  substForm v u (ArithForm.forall_form n f) = if v = n then ArithForm.forall_form n f
else ArithForm.forall_form n (substForm v u f) := rfl

lemma substTerm_var {s : Nat} (v : Nat) (u : ArithTerm s) (n : Nat) :
  substTerm v u (ArithTerm.var n) = if v = n then u else ArithTerm.var n := rfl
```

Lemas de conmutación de sustituciones y alpha-generalización

Se demuestran dos lemas fundamentales sobre la interacción entre sustituciones de variables. El primero establece que dos sustituciones sucesivas con variables intercambiadas se cancelan mutuamente; el segundo garantiza que la condición de libertad se preserva bajo alpha-conversión.

Lema `substTerm_substTerm_self`: Cancelación de sustituciones intercambiadas

Establece que si una variable w no ocurre en un término t, entonces sustituir v por w y luego w por v devuelve el término original:

$$ \forall t \in \text{ArithTerm } s,\ \forall v, w \in \mathbb{N},\ \text{NotOccur } w\ t \implies \text{substTerm } w\ (\text{var } v)\ (\text{substTerm } v\ (\text{var } w)\ t) = t $$

Estructura de la demostración

La prueba procede por inducción estructural sobre t:

- **`var n`:** Se analiza si $n = v$:

- Si $n = v$, la primera sustitución reemplaza $\text{var } v$ por $\text{var } w$, y la segunda reemplaza $\text{var } w$ por $\text{var } v$, resultando en $\text{var } v = \text{var } n$.

- Si $n \neq v$, la primera sustitución no afecta (pues $n \neq v$). La segunda sustitución podría afectar si $n = w$, pero `NotOccur w (var n)` fuerza $w \neq n$, por lo que tampoco afecta. El resultado es $\text{var } n$.

- **Casos base (`const`, `henkin`, `zero`):** Triviales, pues las sustituciones no afectan.

- **Casos recursivos:** Se invierte la hipótesis `NotOccur w t` para obtener las condiciones sobre los subtérminos, y se aplican las hipótesis de inducción.

Intuición

Este lema formaliza la idea de que renombrar una variable v a w (siempre que w sea fresca en t) y luego volver a renombrar w a v es una operación reversible que devuelve el término original. Es la propiedad fundamental que subyace a la alpha-conversión: el cambio de nombre de una variable ligada no altera el significado.

Lema `IsFreeFor_subst_self`: Preservación de la libertad bajo alpha-conversión

Establece que si w no ocurre en φ y v no está ligada en φ, entonces $\text{var } v$ es libre para w en $\varphi[v := \text{var } w]$:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \text{NotOccurForm } w\ \varphi \land \text{NoBoundVar } v\ \varphi \implies \text{IsFreeFor } w\ (\text{var } v)\ (\text{substForm } v\ (\text{var } w)\ \varphi) $$

Estructura de la demostración

La prueba procede por inducción sobre φ:

- **Casos base (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`):** `IsFreeFor` es siempre verdadero para estos constructores.

- **`impl f1 f2`:** Se descomponen las hipótesis y se aplican las hipótesis de inducción a cada subfórmula.

- **`neg f`:** Similar, propagando a la subfórmula.

- **`forall_form n f`:** El caso más delicado. La hipótesis `NoBoundVar v (forall_form n f)` da $v \neq n$ y `NoBoundVar v f`. La hipótesis `NotOccurForm w (forall_form n f)` da $w \neq n$ y `NotOccurForm w f`. La sustitución de v por $\text{var } w$ atraviesa el cuantificador (pues $v \neq n$). Se debe verificar `IsFreeFor w (var v) (forall_form n (substForm v (var w) f))`. Dado que $w \neq n$, se requiere:

  - `NotOccur n (var v)`: cierto porque $v \neq n$.

  - `IsFreeFor w (var v) (substForm v (var w) f)`: cierto por hipótesis de inducción.

Significado

Este lema es técnicamente crucial para los lemas de cambio de variable ligada (`change_var_equivalence_strong`). Garantiza que al realizar una alpha-conversión (renombrar v por w), la nueva variable w cumple la condición de ser libre para cualquier sustitución que involucre a v, lo cual es necesario para poder aplicar las reglas de eliminación del cuantificador universal sobre la fórmula renombrada. La combinación de `substTerm_substTerm_self` e `IsFreeFor_subst_self` proporciona la base sintáctica para todas las operaciones de renombramiento de variables ligadas en el sistema.

```
-- CONMUTACIÓN DE SUSTITUCIONES Y ALPHA-GENERALIZACIÓN

lemma substTerm_substTerm_self {s : Nat} (t : ArithTerm s) (v w : Nat) (hw : NotOccur w t) :
  substTerm w (ArithTerm.var v) (substTerm v (ArithTerm.var w) t) = t := by
 induction t with
 | var n =>
  rw [substTerm_var]
  split_ifs with h1
  · rw [substTerm_var]
    split_ifs with h2
    · subst h1; rfl
    · contradiction
  · rw [substTerm_var]
    split_ifs with _h2
    · cases hw with | var _ h_ne => contradiction
    · rfl
 | const c => rfl
 | henkin c => rfl
 | zero => rfl
 | succ t' ih =>
  cases hw with | succ _ h_sub =>
```

```
    dsimp [substTerm]
    rw [ih h_sub]
  | add t1 t2 ih1 ih2 =>
    cases hw with | add _ _ h1 h2 =>
    dsimp [substTerm]
    rw [ih1 h1, ih2 h2]
  | mul t1 t2 ih1 ih2 =>
    cases hw with | mul _ _ h1 h2 =>
    dsimp [substTerm]
    rw [ih1 h1, ih2 h2]
  | exp t1 t2 ih1 ih2 =>
    cases hw with | exp _ _ h1 h2 =>
    dsimp [substTerm]
    rw [ih1 h1, ih2 h2]
  | pred t' ih =>
    cases hw with | pred _ h_sub =>
    dsimp [substTerm]
    rw [ih h_sub]

lemma IsFreeFor_subst_self {s : Nat} (φ : ArithForm s) (v w : Nat) (hw : NotOccurForm
w φ) (hnv : NoBoundVar v φ) :
    IsFreeFor w (ArithTerm.var v) (substForm v (ArithTerm.var w) φ) := by
  induction φ with
  | bottom => trivial
  | eq _ _ => trivial
  | impl f1 f2 ih1 ih2 =>
    have hw_left : NotOccurForm w f1 := hw.1
    have hw_right : NotOccurForm w f2 := hw.2
```

```
  have hnv_left : NoBoundVar v f1 := hnv.1
  have hnv_right : NoBoundVar v f2 := hnv.2
  rw [substForm_impl, IsFreeFor_impl]
  exact ⟨ih1 hw_left hnv_left, ih2 hw_right hnv_right⟩
| neg f ih =>
  have hw_inner : NotOccurForm w f := hw
  have hnv_inner : NoBoundVar v f := hnv
  rw [substForm_neg, IsFreeFor_neg]
  exact ih hw_inner hnv_inner
| forall_form n f ih =>
  have hw_left : w ≠ n := hw.1
  have hw_right : NotOccurForm w f := hw.2
  have hnv_ne : v ≠ n := hnv.1
  have hnv_inner : NoBoundVar v f := hnv.2
  rw [substForm_forall_form]
  rw [if_neg hnv_ne]
  rw [IsFreeFor_forall_form]
  by_cases h_w : w = n
  · exact (hw_left h_w).elim
  · rw [if_neg h_w]
    constructor
    · exact NotOccur.var v (Ne.symm hnv_ne)
    · exact ih hw_right hnv_inner
| provable_op _ _ => trivial
| true_op _ _ => trivial
| list_nth_eq _ _ _ => trivial
```

Lema `substForm_substForm_self` y Teorema `alpha_generalize`

Se demuestra el lema de cancelación de sustituciones intercambiadas para fórmulas y el teorema de generalización alpha, que permite cambiar el nombre de una variable ligada en una deducción.

Lema `substForm_substForm_self`: Cancelación de sustituciones intercambiadas en fórmulas

Extiende `substTerm_substTerm_self` al nivel de fórmulas: si w no ocurre en φ, entonces sustituir v por $\text{var } w$ y luego w por $\text{var } v$ devuelve la fórmula original:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \text{NotOccurForm } w\ \varphi \implies \text{substForm } w\ (\text{var } v)\ (\text{substForm } v\ (\text{var } w)\ \varphi) = \varphi $$

Estructura de la demostración

La prueba procede por inducción sobre φ, utilizando `substTerm_substTerm_self` para los casos que involucran términos y las hipótesis de inducción para los casos recursivos. El caso `forall_form n f` es el más delicado:

- Si $v = n$, la sustitución interna no afecta al cuantificador, y la sustitución externa sobre w (con $w \neq n$ por `NotOccurForm`) tampoco afecta al cuantificador. Se utiliza `substForm_fresh` para resolver la sustitución sobre f.

- Si $v \neq n$ y $w \neq n$, ambas sustituciones atraviesan el cuantificador y la hipótesis de inducción se aplica a f.

Teorema `alpha_generalize`: Generalización alpha o cambio de variable ligada

Establece una regla derivada fundamental: si una fórmula $\neg \psi[v := \text{var } w]$ es demostrable en un contexto donde w es fresca, entonces $\forall v\ \neg \psi$ también es demostrable. Formalmente:

$$ \Gamma \vdash \neg (\psi[v := \text{var } w]) \implies \Gamma \vdash \forall v\ \neg \psi $$

bajo las hipótesis:

- w no es libre en Γ (`NotFreeInSet w Γ`)

- w no ocurre en ψ (`NotOccurForm w ψ`)

- v no es libre en Γ (`NotFreeInSet v Γ`)

- v no está ligada en ψ (`NoBoundVar v ψ`)

Estructura de la demostración

1. Generalización sobre w: Dado que w no es libre en Γ, se aplica `forall_intro` para obtener $\Gamma \vdash \forall w\ \neg(\psi[v := \text{var } w])$.

2. Instanciación con v: Se verifica que $\text{var } v$ es libre para w en $\neg(\psi[v := \text{var } w])$ usando `IsFreeFor_subst_self` (que requiere precisamente `NotOccurForm w ψ` y `NoBoundVar v ψ`). Se aplica `forall_elim` para obtener $\Gamma \vdash (\neg(\psi[v := \text{var } w]))[w := \text{var } v]$.

3. Simplificación: Por `substForm_substForm_self`, la doble sustitución colapsa a $\neg \psi$. La fórmula resultante es exactamente $\Gamma \vdash \neg \psi$.

4. Generalización sobre v: Dado que v no es libre en Γ, se aplica `forall_intro` para obtener $\Gamma \vdash \forall v\ \neg \psi$.

Significado

`alpha_generalize` es una de las reglas derivadas más importantes del sistema deductivo. Captura el principio de que si se puede demostrar $\neg \psi$ con una variable fresca w en lugar de v, entonces se puede generalizar a $\forall v\ \neg \psi$. Esto es esencialmente el Teorema de Generalización combinado con alpha-conversión: el cambio de variable ligada es lícito siempre que la nueva variable sea fresca en el contexto y en la fórmula.

Este teorema es utilizado profusamente en las demostraciones de inducción interna (como las de `add_comm_Δ`, `succ_mul_Δ`, etc.) y en la construcción de testigos de Henkin, donde se necesita pasar de una demostración que involucra una variable fresca a una afirmación universalmente cuantificada. La combinación de `substForm_substForm_self` e `IsFreeFor_subst_self` proporciona la justificación sintáctica completa para la operación de renombramiento de variables ligadas, que es ubicua en la manipulación de fórmulas con cuantificadores.

```
lemma substForm_substForm_self {s : Nat} (φ : ArithForm s) (v w : Nat) (hw : NotOccurForm w φ) :
  substForm w (ArithTerm.var v) (substForm v (ArithTerm.var w) φ) = φ := by
 induction φ with
 | bottom => rfl
 | eq t1 t2 =>
  have hw_left : NotOccur w t1 := hw.left
  have hw_right : NotOccur w t2 := hw.right
  rw [substForm_eq, substForm_eq]
  rw [substTerm_substTerm_self t1 v w hw_left, substTerm_substTerm_self t2 v w hw_right]
 | impl f1 f2 ih1 ih2 =>
  have hw_left : NotOccurForm w f1 := hw.left
  have hw_right : NotOccurForm w f2 := hw.right
  rw [substForm_impl, substForm_impl]
  rw [ih1 hw_left, ih2 hw_right]
 | neg f ih =>
  have hw_inner : NotOccurForm w f := hw
  rw [substForm_neg, substForm_neg]
  rw [ih hw_inner]
 | forall_form n f ih =>
  have hw_left : w ≠ n := hw.left
  have hw_right : NotOccurForm w f := hw.right
  rw [substForm_forall_form]
  by_cases h_v : v = n
  · rw [if_pos h_v]
    rw [substForm_forall_form]
    rw [if_neg hw_left]
    rw [substForm_fresh f w (ArithTerm.var v) hw_right]
```

```
    · rw [if_neg h_v]
      rw [substForm_forall_form]
      rw [if_neg hw_left]
      rw [ih hw_right]
  | provable_op heq t =>
    have hw_inner : NotOccur w t := hw
    rw [substForm_provable_op, substForm_provable_op]
    rw [substTerm_substTerm_self t v w hw_inner]
  | true_op heq t =>
    have hw_inner : NotOccur w t := hw
    rw [substForm_true_op, substForm_true_op]
    rw [substTerm_substTerm_self t v w hw_inner]
  | list_nth_eq t1 t2 t3 =>
    have hw_t1 : NotOccur w t1 := hw.left
    have hw_t2 : NotOccur w t2 := hw.right.left
    have hw_t3 : NotOccur w t3 := hw.right.right
    rw [substForm_list_nth_eq, substForm_list_nth_eq]
    rw [substTerm_substTerm_self t1 v w hw_t1, substTerm_substTerm_self t2 v w
hw_t2, substTerm_substTerm_self t3 v w hw_t3]


theorem alpha_generalize {s : Nat} (Γ : Set (ArithForm s)) (v w : Nat) (ψ : ArithForm s)
    (h_fresh_Γ : NotFreeInSet w Γ) (h_fresh_ψ : NotOccurForm w ψ)
    (h_fresh_v : NotFreeInSet v Γ)
    (h_v_bound : NoBoundVar v ψ)
    (h_deriv : SPA_Deduction_Internal s Γ (ArithForm.neg (substForm v (ArithTerm.var
w) ψ))) :
    SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.neg ψ)) := by
  have h1 : SPA_Deduction_Internal s Γ (ArithForm.forall_form w (ArithForm.neg
(substForm v (ArithTerm.var w) ψ))) :=
```

```
  SPA_Deduction_Internal.forall_intro Γ w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)) h_deriv h_fresh_Γ

 have h_free : IsFreeFor w (ArithTerm.var v) (ArithForm.neg (substForm v (ArithTerm.var w) ψ)) := by

  rw [IsFreeFor_neg]

  exact IsFreeFor_subst_self ψ v w h_fresh_ψ h_v_bound

 have h2 : SPA_Deduction_Internal s Γ (substForm w (ArithTerm.var v) (ArithForm.neg (substForm v (ArithTerm.var w) ψ))) :=

  SPA_Deduction_Internal.forall_elim Γ w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)) (ArithTerm.var v) h1 h_free

 have h_eq : substForm w (ArithTerm.var v) (ArithForm.neg (substForm v (ArithTerm.var w) ψ)) = ArithForm.neg ψ := by

  change ArithForm.neg (substForm w (ArithTerm.var v) (substForm v (ArithTerm.var w) ψ)) = ArithForm.neg ψ

  rw [substForm_substForm_self ψ v w h_fresh_ψ]

 rw [h_eq] at h2

 exact SPA_Deduction_Internal.forall_intro Γ v (ArithForm.neg ψ) h2 h_fresh_v
```

Lema `max_var_list_ge` y configuración de la secuencia de Henkin

Se completa la infraestructura para la construcción de Henkin con un lema sobre cotas de variables en listas y las definiciones para detectar y decodificar fórmulas existenciales.

Lema `max_var_list_ge`: Cota para una fórmula en una lista

Establece que si una fórmula φ pertenece a una lista L, entonces el máximo de variables de φ está acotado por el máximo de variables de la lista:

$$ \forall L \in \text{List}(\text{ArithForm } s),\ \forall \varphi \in \text{ArithForm } s,\ \varphi \in L \implies \text{max_var_form}(\varphi) \le \text{max_var_list}(L) $$

Estructura de la demostración

Por inducción sobre L:

- **Lista vacía:** No hay fórmulas, la premisa es contradictoria.

- **Lista con cabeza ψ y cola L':** Si $\varphi = \psi$, entonces $\text{max_var_form}(\varphi) = \text{max_var_form}(\psi) \le \max(\text{max_var_form}(\psi), \text{max_var_list}(L'))$. Si $\varphi \in L'$, por hipótesis de inducción $\text{max_var_form}(\varphi) \le \text{max_var_list}(L')$, y este último está acotado por el máximo con $\text{max_var_form}(\psi)$.

Intuición

Este lema es utilizado en la construcción de testigos de Henkin para garantizar que las nuevas variables elegidas (con índice mayor que `max_var_list`) son frescas respecto a todas las fórmulas acumuladas hasta el momento.

Funciones para la secuencia de Henkin

`getExistentialComponents`: Detecta si una fórmula tiene la forma $\neg \forall v \neg \psi$ (es decir, $\exists v\ \psi$) y extrae la variable v y la fórmula ψ:

$$ \text{getExistentialComponents}(\varphi) = \begin{cases} \text{some}(v, \psi) & \text{si } \varphi = \neg \forall v \neg \psi \\ \text{none} & \text{en otro caso} \end{cases} $$

`defaultExists`: Fórmula existencial por defecto utilizada cuando la enumeración no produce una fórmula válida:

$$ \text{defaultExists}(s) := \neg \forall v_0 \neg \bot = \exists v_0\ \neg \bot $$

`decodeExists`: Decodifica el n-ésimo elemento de la enumeración de fórmulas, verificando si es una fórmula existencial. Si lo es, la devuelve; en caso contrario, devuelve `defaultExists`:

$$ \text{decodeExists}(n) = \begin{cases} \varphi & \text{si } \text{decode}(n) = \text{some } \varphi \text{ y } \varphi \text{ es existencial} \\ \text{defaultExists} & \text{en otro caso} \end{cases} $$

Lema `decodeExists_eq_defaultExists_of_getExistentialComponents_none`

Establece que si `getExistentialComponents` devuelve `none` para la fórmula decodificada, entonces `decodeExists` devuelve `defaultExists`:

$$ \text{getExistentialComponents}(\text{decodeExists } m) = \text{none} \implies \text{decodeExists } m = \text{defaultExists } 0 $$

La demostración analiza los casos de la decodificación: si la decodificación falla, se devuelve `defaultExists`; si tiene éxito pero la fórmula no es existencial, también se

devuelve `defaultExists`; si fuera existencial, `getExistentialComponents` devolvería `some`, contradiciendo la hipótesis.

Significado

Estas definiciones y lemas configuran la maquinaria para la construcción de la secuencia de Henkin. La idea es enumerar todas las fórmulas del lenguaje (usando `Encodable`) y, para cada una que sea existencial, añadir un axioma de testigo con una constante de Henkin fresca. `decodeExists` garantiza que en cada paso de la enumeración se procesa una fórmula existencial (ya sea genuina o la fórmula por defecto), lo que permite un tratamiento uniforme en la definición de `GammaWitnessesList` y `GammaSeq`. La fórmula por defecto es una tautología existencial ($\exists v_0\ \neg \bot$) que no afecta la consistencia y asegura que siempre hay algo que procesar en cada paso.

```
lemma max_var_list_ge (L : List (ArithForm 0)) (φ : ArithForm 0) (h : φ ∈ L) : max_var_form φ ≤ max_var_list L := by
  induction L with
  | nil => contradiction
  | cons ψ L' ih =>
    unfold max_var_list
    cases h
    · exact Nat.le_max_left _ _
    · rename_i h'
      exact Nat.le_trans (ih h') (Nat.le_max_right _ _)
```

```
--
================================================================
-- CONFIGURACIÓN DE LA SECUENCIA DE HENKIN Y TESTIGOS
--
================================================================

def getExistentialComponents {s : Nat} (φ : ArithForm s) : Option (Nat × ArithForm s)
:=
  match φ with
  | ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ)) => some (v, ψ)
  | _ => none

def defaultExists (s : Nat) : ArithForm s :=
  ArithForm.neg (ArithForm.forall_form 0 (ArithForm.neg ArithForm.bottom))

def decodeExists {s : Nat} [Encodable (ArithForm s)] (n : Nat) : ArithForm s :=
  match (Encodable.decode n : Option (ArithForm s)) with
  | some φ =>
    match (getExistentialComponents (s := s) φ : Option (Nat × ArithForm s)) with
    | some _ => φ
    | none => defaultExists s
  | none => defaultExists s




lemma decodeExists_eq_defaultExists_of_getExistentialComponents_none (m : Nat)
    (h : getExistentialComponents (s := 0) (decodeExists (s := 0) m) = none) :
```

```
    decodeExists (s := 0) m = defaultExists 0 := by
  unfold decodeExists at h ⊢
  generalize h_dec : @Encodable.decode (ArithForm 0) (ArithForm_encodable 0) m = x
at h ⊢
  cases x with
  | none =>
    rfl
  | some φ =>
    dsimp only at h ⊢
    generalize h_dec' : getExistentialComponents (s := 0) φ = y at h ⊢
    cases y with
    | some val =>
      change getExistentialComponents (s := 0) φ = none at h
      rw [h_dec'] at h
      contradiction
    | none =>
      rfl
```

Lema de decodificación existencial e infraestructura para constantes de Henkin frescas

Se completa la configuración de la secuencia de Henkin con un lema sobre la extracción de componentes existenciales y las funciones para generar constantes de Henkin frescas basadas en cotas superiores.

Lema `decodeExists_eq_of_getExistentialComponents_some`: Extracción de fórmula existencial

Establece que si `getExistentialComponents` devuelve un par (v, ψ) para la fórmula decodificada, entonces dicha fórmula es exactamente $\neg \forall v \neg \psi$ (es decir, la representación de $\exists v\ \psi$):

$$ \text{getExistentialComponents}(\text{decodeExists } m) = \text{some}(v, \psi) \implies \text{decodeExists } m = \neg \forall v \neg \psi $$

La demostración generaliza la expresión `decodeExists m`, despliega la definición de `getExistentialComponents`, y por inversión del patrón `some (v, ψ)` obtiene que la fórmula debe ser de la forma `neg (forall_form v (neg ψ))`.

Funciones para constantes de Henkin frescas

Se definen cuatro funciones que calculan cotas superiores para los índices de constantes de Henkin que aparecen en términos, fórmulas y listas de fórmulas:

`maxHenkinTerm`: Calcula el máximo índice de constante de Henkin en un término:

$$ \text{maxHenkinTerm} : \text{ArithTerm } s \to \mathbb{N} $$

Para `henkin c` devuelve `c.index`; para los demás constructores base devuelve 0; para constructores compuestos toma el máximo de los subtérminos.

`maxHenkinForm`: Calcula el máximo índice de constante de Henkin en una fórmula, propagándose a través de todos los constructores de fórmulas y utilizando `maxHenkinTerm` para los términos que aparecen en igualdades y operadores estratificados.

`maxHenkinList`: Extiende el cálculo a listas de fórmulas, tomando el máximo de `maxHenkinForm` sobre todos los elementos.

`freshConst`: Genera una nueva constante de Henkin fresca respecto a una lista de fórmulas L, con índice estrictamente mayor que todas las constantes que aparecen en L:

$$ \text{freshConst}(L) := \langle \text{maxHenkinList}(L) + 1 \rangle $$

Intuición y necesidad

Estas funciones son el análogo para constantes de Henkin de lo que `max_var_term`, `max_var_form` y `max_var_list` son para variables. Durante la construcción de la secuencia de Henkin, cuando se procesa una fórmula existencial $\exists v\ \psi$, se necesita generar una nueva constante de Henkin que no haya sido utilizada previamente. `freshConst` garantiza precisamente esto: el índice de la nueva constante es estrictamente mayor que cualquier índice de constante que aparezca en las fórmulas testigo acumuladas hasta el momento.

La combinación de frescura para variables (mediante `max_var_list`) y frescura para constantes de Henkin (mediante `maxHenkinList`) asegura que los nuevos símbolos introducidos en cada paso de la construcción no colisionan con ninguno de los símbolos ya utilizados, lo cual es indispensable para preservar la consistencia durante la extensión de la teoría. Esta técnica de "frescura por cota superior" unifica el

tratamiento de variables y constantes, y es una de las claves técnicas que hacen viable la construcción de Henkin-Lindenbaum en el sistema SPA.

```
lemma decodeExists_eq_of_getExistentialComponents_some (m : Nat) (v : Nat) (ψ : ArithForm 0)
    (h_dec : getExistentialComponents (decodeExists (s := 0) m) = some (v, ψ)) :
    decodeExists (s := 0) m = ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ)) := by
  generalize h_x : decodeExists (s := 0) m = χ at h_dec ⊢
  unfold getExistentialComponents at h_dec
  split at h_dec
  · rename_i v' ψ'
    injection h_dec with h_pair
    injection h_pair with h_v h_ψ
    subst h_v h_ψ
    rfl
  · contradiction
```

```
def maxHenkinTerm {s : Nat} : ArithTerm s → Nat
```

```
  | ArithTerm.var _ => 0
  | ArithTerm.const _ => 0
  | ArithTerm.henkin c => c.index
  | ArithTerm.zero => 0
  | ArithTerm.succ t => maxHenkinTerm t
  | ArithTerm.add t1 t2 => Nat.max (maxHenkinTerm t1) (maxHenkinTerm t2)
  | ArithTerm.mul t1 t2 => Nat.max (maxHenkinTerm t1) (maxHenkinTerm t2)
  | ArithTerm.exp t1 t2 => Nat.max (maxHenkinTerm t1) (maxHenkinTerm t2)
  | ArithTerm.pred t => maxHenkinTerm t

def maxHenkinForm {s : Nat} : ArithForm s → Nat
  | ArithForm.bottom => 0
  | ArithForm.eq t1 t2 => Nat.max (maxHenkinTerm t1) (maxHenkinTerm t2)
  | ArithForm.impl f1 f2 => Nat.max (maxHenkinForm f1) (maxHenkinForm f2)
  | ArithForm.neg f => maxHenkinForm f
  | ArithForm.forall_form _ f => maxHenkinForm f
  | ArithForm.provable_op _ t => maxHenkinTerm t
  | ArithForm.true_op _ t => maxHenkinTerm t
  | ArithForm.list_nth_eq t1 t2 t3 => Nat.max (maxHenkinTerm t1) (Nat.max
(maxHenkinTerm t2) (maxHenkinTerm t3))

def maxHenkinList {s : Nat} : List (ArithForm s) → Nat
  | [] => 0
  | f :: fs => Nat.max (maxHenkinForm f) (maxHenkinList fs)

def freshConst {s : Nat} (L : List (ArithForm s)) : HenkinConst :=
  ⟨maxHenkinList L + 1⟩
```

Lemas de soporte estructural de frescura para rectificación

Se demuestran cuatro lemas que establecen propiedades fundamentales sobre la preservación de la frescura y la no ocurrencia bajo operaciones de sustitución y cambio de variable. Estos lemas son la base técnica para la "rectificación" de fórmulas, es decir, el proceso de asegurar que ciertas variables no aparecen ligadas en una fórmula.

Lema `NotOccur_substTerm_self_var_global`: No ocurrencia tras auto-sustitución

Establece que si $v \neq w$, entonces v no ocurre en el resultado de sustituir v por $\text{var } w$ en cualquier término t:

$$ \forall t \in \text{ArithTerm } s,\ \forall v, w \in \mathbb{N},\ v \neq w \implies \text{NotOccur } v\ (\text{substTerm } v\ (\text{var } w)\ t) $$

Intuición

Al sustituir la variable v por $\text{var } w$ (con $w \neq v$), todas las ocurrencias de v en t son reemplazadas por w. Dado que $w \neq v$, el resultado no contiene ninguna ocurrencia de v. La demostración es por inducción estructural sobre t, analizando el caso `var n` según si $n = v$ (se convierte en $\text{var } w$, que no es v) o $n \neq v$ (se preserva, y $n \neq v$).

Lema `NotFreeForm_subst_var_self_global`: Variable no libre tras auto-sustitución

Extiende el lema anterior al nivel de fórmulas: si $v \neq w$, entonces v no es libre en $\psi[v := \text{var } w]$:

$$ \forall \psi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ v \neq w \implies \text{NotFreeForm } v\ (\text{substForm } v\ (\text{var } w)\ \psi) $$

Estructura de la demostración

Por inducción sobre ψ, utilizando `NotOccur_substTerm_self_var_global` para los casos con términos. El caso `forall_form n f` distingue si $v = n$: si coinciden, la sustitución no penetra y v está ligada, por lo que no es libre; si no coinciden, la sustitución se propaga y la hipótesis de inducción garantiza que v no es libre en el resultado.

Lema `NotFreeForm_of_NotOccurForm_global`: No ocurrencia implica no libertad

Establece la relación jerárquica entre los dos predicados: si una variable no ocurre en absoluto en una fórmula, entonces no es libre:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \text{NotOccurForm } v\ \varphi \implies \text{NotFreeForm } v\ \varphi $$

La demostración es por inducción sobre φ, verificando que cada condición de `NotOccurForm` implica la condición correspondiente de `NotFreeForm`. El caso `forall_form n f` es el único donde los predicados difieren: `NotOccurForm` requiere $v \neq n$ y que v no ocurra en f, mientras que `NotFreeForm` permite que $v = n$

(en cuyo caso v está ligada y por tanto no es libre). La implicación se verifica en ambos subcasos.

Lema `NoBoundVar_of_NotOccurForm_global`: No ocurrencia implica no ligadura

Análogo al anterior: si una variable no ocurre en absoluto en una fórmula, entonces no está ligada en ella:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall w \in \mathbb{N},\ \text{NotOccurForm } w\ \varphi \implies \text{NoBoundVar } w\ \varphi $$

La demostración es directa por inducción, ya que `NotOccurForm` para `forall_form` ya incluye la condición $w \neq n$.

Lema `NoBoundVar_substForm_general_global`: Preservación de no ligadura bajo sustitución

Establece que la propiedad de no estar ligada se preserva bajo cualquier sustitución:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \forall z \in \mathbb{N},\ \text{NoBoundVar } z\ \varphi \implies \text{NoBoundVar } z\ (\text{substForm } v\ t\ \varphi) $$

La demostración es por inducción sobre φ, verificando que la sustitución no introduce nuevas ligaduras (los cuantificadores existentes no se modifican, solo se modifica el cuerpo). El caso `forall_form n f` distingue si la sustitución atraviesa o no el cuantificador; en ambos subcasos, la condición $z \neq n$ se preserva.

Significado conjunto

Estos lemas constituyen la infraestructura para garantizar que las operaciones de "rectificación" (renombrar variables ligadas para evitar conflictos) son seguras y preservan las propiedades de frescura necesarias. Son utilizados en los lemas de cambio de variable ligada y en la construcción de testigos de Henkin, donde se necesita asegurar que las variables elegidas como frescas no aparecen libres ni ligadas en las fórmulas relevantes, y que estas propiedades se mantienen a lo largo de las transformaciones sintácticas. La distinción entre `NotOccurForm`, `NotFreeForm` y `NoBoundVar` permite un control fino sobre las diferentes nociones de "no aparecer" en una fórmula, cada una relevante para diferentes reglas de inferencia.

```
--
============================================================================
-- CONFIGURACIÓN DE LA SECUENCIA DE HENKIN Y TESTIGOS (SITUACIÓN FINAL)
--
============================================================================
```

```
--
============================================================================
-- RECTIFICACIÓN DE FÓRMULAS (Garantiza NoBoundVar para toda variable)
--
============================================================================
```

```
--
================================================================
-- LEMAS DE SOPORTE ESTRUCTURAL DE FRESCURA PARA RECTIFICACIÓN
--
================================================================

lemma NotOccur_substTerm_self_var_global {s : Nat} (t : ArithTerm s) (v w : Nat)
(h_vw : v ≠ w) :
  NotOccur v (substTerm v (ArithTerm.var w) t) := by
 induction t with
 | var n =>
  unfold substTerm
  split_ifs with h_vn
  · exact NotOccur.var w h_vw
  · exact NotOccur.var n h_vn
 | const c => exact NotOccur.const c
 | henkin c => exact NotOccur.henkin c
 | zero => exact NotOccur.zero
 | succ t' ih => exact NotOccur.succ _ ih
 | add t1 t2 ih1 ih2 => exact NotOccur.add _ _ ih1 ih2
 | mul t1 t2 ih1 ih2 => exact NotOccur.mul _ _ ih1 ih2
 | exp t1 t2 ih1 ih2 => exact NotOccur.exp _ _ ih1 ih2
 | pred t' ih => exact NotOccur.pred _ ih

lemma NotFreeForm_subst_var_self_global {s : Nat} (ψ : ArithForm s) (v w : Nat)
(h_vw : v ≠ w) :
  NotFreeForm v (substForm v (ArithTerm.var w : ArithTerm s) ψ) := by
 induction ψ with
```

```
  | bottom => trivial
  | eq t1 t2 =>
    dsimp [substForm, NotFreeForm]
    exact ⟨NotOccur_substTerm_self_var_global t1 v w h_vw, NotOccur_substTerm_self_var_global t2 v w h_vw⟩
  | impl f1 f2 ih1 ih2 =>
    dsimp [substForm, NotFreeForm]
    exact ⟨ih1, ih2⟩
  | neg f ih =>
    dsimp [substForm, NotFreeForm]
    exact ih
  | forall_form n f ih =>
    dsimp [substForm]
    by_cases h_vn : v = n
    · rw [if_pos h_vn]
      dsimp [NotFreeForm]
      rw [if_pos h_vn]
      trivial
    · rw [if_neg h_vn]
      dsimp [NotFreeForm]
      rw [if_neg h_vn]
      exact ih
  | provable_op h t =>
    dsimp [substForm, NotFreeForm]
    exact NotOccur_substTerm_self_var_global t v w h_vw
  | true_op h t =>
    dsimp [substForm, NotFreeForm]
    exact NotOccur_substTerm_self_var_global t v w h_vw
```

```
  | list_nth_eq t1 t2 t3 =>
    dsimp [substForm, NotFreeForm]
    exact ⟨NotOccur_substTerm_self_var_global t1 v w h_vw, NotOccur_substTerm_self_var_global t2 v w h_vw, NotOccur_substTerm_self_var_global t3 v w h_vw⟩

lemma NotFreeForm_of_NotOccurForm_global {s : Nat} (v : Nat) (φ : ArithForm s) (h : NotOccurForm v φ) : NotFreeForm v φ := by
  induction φ generalizing v with
  | bottom => trivial
  | eq t1 t2 => exact h
  | impl f1 f2 ih1 ih2 =>
    exact ⟨ih1 v h.1, ih2 v h.2⟩
  | neg f ih =>
    exact ih v h
  | forall_form n f ih =>
    dsimp [NotFreeForm]
    by_cases h_vn : v = n
    · rw [if_pos h_vn]
      exact True.intro
    · rw [if_neg h_vn]
      exact ih v h.2
  | provable_op _ _ => exact h
  | true_op _ _ => exact h
  | list_nth_eq _ _ _ => exact h

lemma NoBoundVar_of_NotOccurForm_global {s : Nat} (w : Nat) (φ : ArithForm s) (h : NotOccurForm w φ) :
    NoBoundVar w φ := by
  induction φ with
```

```
  | bottom => trivial
  | eq _ _ => trivial
  | impl f1 f2 ih1 ih2 =>
    exact ⟨ih1 h.1, ih2 h.2⟩
  | neg f ih =>
    exact ih h
  | forall_form n f ih =>
    exact ⟨h.1, ih h.2⟩
  | provable_op _ _ => trivial
  | true_op _ _ => trivial
  | list_nth_eq _ _ _ => trivial

lemma NoBoundVar_substForm_general_global {s : Nat} (φ : ArithForm s) (v : Nat) (t :
ArithTerm s) (z : Nat) (h : NoBoundVar z φ) :
    NoBoundVar z (substForm v t φ) := by
  induction φ generalizing v t z with
  | bottom => trivial
  | eq _ _ => trivial
  | impl f1 f2 ih1 ih2 =>
    dsimp [substForm, NoBoundVar] at h ⊢
    exact ⟨ih1 v t z h.1, ih2 v t z h.2⟩
  | neg f ih =>
    dsimp [substForm, NoBoundVar] at h ⊢
    exact ih v t z h
  | forall_form n f ih =>
    dsimp [substForm]
    split_ifs with h_vn
    · dsimp [NoBoundVar] at h ⊢
```

```
    exact h
  · dsimp [NoBoundVar] at h ⊢
    exact ⟨h.1, ih v t z h.2⟩
 | provable_op _ _ => trivial
 | true_op _ _ => trivial
 | list_nth_eq _ _ _ => trivial
```

Lema `IsFreeFor_var_self_global` y lemas auxiliares de deducción

Se completan las herramientas básicas de manipulación deductiva con un lema sobre la libertad de una variable respecto a sí misma y dos lemas sobre deducciones en contexto vacío.

Lema `IsFreeFor_var_self_global`: Una variable es libre para sí misma en cualquier fórmula

Establece que para cualquier fórmula φ y cualquier variable v, el término $\text{var } v$ es libre para v en φ:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \text{IsFreeFor } v\ (\text{var } v)\ \varphi $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos base (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`):** `IsFreeFor` es siempre verdadero.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Si $v = n$, la sustitución no penetra y la condición es trivial. Si $v \neq n$, se requiere `NotOccur n (var v)` (que se cumple porque $v \neq n$) y la hipótesis de inducción para f.

Intuición

Este lema refleja el hecho de que sustituir una variable por sí misma nunca causa captura: no hay variables libres nuevas que puedan ser atrapadas por cuantificadores. Es un caso particular de la condición de libertad que se verifica trivialmente.

Lema `impl_self`: Identidad de la implicación

Establece que $\varphi \to \varphi$ es demostrable en cualquier contexto Γ:

$$ \forall \Gamma \subseteq \text{ArithForm } s,\ \forall \varphi \in \text{ArithForm } s,\ \Gamma \vdash \varphi \to \varphi $$

La demostración es directa: por `imp_intro`, basta demostrar φ en el contexto extendido con φ, lo cual es inmediato por `hyp`.

Lema `subst_empty_ctx`: Sustitución en contexto vacío

Establece que si una fórmula es demostrable en contexto vacío, entonces cualquier instancia de sustitución también lo es, siempre que la sustitución sea segura:

$$ \forall v \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \forall \varphi \in \text{ArithForm } s,\ \emptyset \vdash \varphi \land \text{IsFreeFor } v\ t\ \varphi \implies \emptyset \vdash \varphi[v := t] $$

Estructura de la demostración

1. Dado que $\emptyset \vdash \varphi$ y el conjunto vacío no contiene fórmulas, v no es libre en el contexto (trivialmente). Se aplica `forall_intro` para obtener $\emptyset \vdash \forall v\ \varphi$.

2. Dado que t es libre para v en φ, se aplica `forall_elim` para obtener $\emptyset \vdash \varphi[v := t]$.

Significado

Este lema es una versión del principio de sustitución para teoremas lógicos: si una fórmula es un teorema (demostrable sin hipótesis), entonces cualquier instancia de sustitución también es un teorema. Es utilizado en las demostraciones de inducción interna, donde las fórmulas de inducción se demuestran primero en contexto vacío y luego se instancian con términos concretos. La restricción al contexto vacío es esencial porque `forall_intro` requiere que la variable generalizada no sea libre en las hipótesis, lo cual se satisface automáticamente cuando no hay hipótesis.

```
lemma IsFreeFor_var_self_global {s : Nat} (φ : ArithForm s) (v : Nat) : IsFreeFor v (ArithTerm.var v) φ := by
  induction φ generalizing v with
  | bottom => trivial
  | eq t1 t2 => trivial
  | impl f1 f2 ih1 ih2 => exact ⟨ih1 v, ih2 v⟩
  | neg f ih => exact ih v
  | forall_form n f ih =>
    dsimp [IsFreeFor]
    by_cases h : v = n
    · rw [if_pos h]
      exact True.intro
    · rw [if_neg h]
      exact ⟨NotOccur.var v (Ne.symm h), ih v⟩
  | provable_op _ _ => trivial
  | true_op _ _ => trivial
  | list_nth_eq _ _ _ => trivial
```

```
--
================================================================
-- LEMAS DE CAMBIO DE VARIABLE GENERALES EN CONTEXTO VACÍO
--
================================================================




--
================================================================
-- OPERACIONES DE SUSTITUCIÓN DE IDENTIDAD Y EN CONTEXTO VACÍO
--
================================================================


lemma impl_self {s : Nat} (Γ : Set (ArithForm s)) (φ : ArithForm s) :
SPA_Deduction_Internal s Γ (ArithForm.impl φ φ) := by
  apply SPA_Deduction_Internal.imp_intro
  apply SPA_Deduction_Internal.hyp
  exact Or.inl rfl


lemma subst_empty_ctx {s : Nat} (v : Nat) (t : ArithTerm s) (φ : ArithForm s)
    (h : SPA_Deduction_Internal s ∅ φ) (h_free : IsFreeFor v t φ) :
    SPA_Deduction_Internal s ∅ (substForm v t φ) := by
  have h1 : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ) := by
    apply SPA_Deduction_Internal.forall_intro ∅ v φ h
    intro ψ hψ
```

```
  nomatch hψ
exact SPA_Deduction_Internal.forall_elim ∅ v φ t h1 h_free
```

Lemas de identidad de sustitución y construcción de la secuencia de Henkin

Se demuestran dos lemas que establecen que la sustitución de una variable por sí misma es la identidad, y se presentan las definiciones centrales de la construcción de Henkin para el nivel base $s = 0$.

Lema `substTerm_self_var`: Sustituir una variable por sí misma no modifica el término

Establece que para cualquier término t y cualquier variable v, sustituir v por $\text{var } v$ deja t inalterado:

$$ \forall t \in \text{ArithTerm } s,\ \forall v \in \mathbb{N},\ \text{substTerm } v\ (\text{var } v)\ t = t $$

La demostración es por inducción estructural sobre t. Para `var n`, si $v = n$ se reemplaza por $\text{var } v$ (que es lo mismo); si $v \neq n$, no se modifica. Los demás casos se resuelven por las hipótesis de inducción.

Lema `substForm_self_var`: Sustituir una variable por sí misma no modifica la fórmula

Extiende el lema anterior a fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \text{substForm } v\ (\text{var } v)\ \varphi = \varphi $$

La demostración es por inducción sobre φ, utilizando `substTerm_self_var` para los casos con términos y las hipótesis de inducción para los casos recursivos. En `forall_form n f`, si $v = n$ la sustitución no penetra; si $v \neq n$, se aplica la hipótesis de inducción.

Construcción de la secuencia de Henkin para $s = 0$

Se definen las dos funciones principales que implementan la construcción de Henkin para el nivel base:

`GammaWitnessesList`: Lista acumulativa de axiomas de testigo añadidos durante la enumeración. En cada paso $m+1$:

1. Se decodifica la m-ésima fórmula mediante `decodeExists`.

2. Si es una fórmula existencial $\exists v\ \psi$ (extraída por `getExistentialComponents`):

  - Se calcula una variable fresca w usando `max_var_list` y `max_var_form` para evitar colisiones.

  - Se construye la fórmula existencial fresca `freshEx` $:= \neg \forall w \neg (\psi[v := \text{var } w])$.

- Se genera una constante de Henkin fresca c usando `maxHenkinList` y `maxHenkinForm`.

- Se añade el axioma de testigo: $\text{freshEx} \to \psi[v := \text{henkin } c]$.

3. Si no es existencial, la lista no se modifica.

`GammaSeq`: Secuencia creciente de conjuntos de fórmulas. En cada paso, se inserta el axioma de testigo correspondiente (si la fórmula es existencial) en el conjunto acumulado del paso anterior.

Intuición y necesidad

Esta construcción es el corazón del método de Henkin para demostrar la completitud. La idea es expandir el lenguaje con constantes de Henkin que actúan como testigos para las fórmulas existenciales, garantizando que toda fórmula existencial demostrable tenga una instancia cerrada que la satisface. La frescura de las variables y constantes introducidas (asegurada por tomar índices mayores que todos los existentes) es crucial para preservar la consistencia durante la expansión. La construcción itera sobre una enumeración de todas las fórmulas del lenguaje (provista por `Encodable`), procesando una por una y añadiendo los axiomas de testigo necesarios. El límite de esta secuencia será una teoría de Henkin, existencialmente saturada.

--
==========

-- TEOREMA PRINCIPAL DE EQUIVALENCIA DEDUCTIVA DE RECTIFICACIÓN

--
==========

```
lemma substTerm_self_var {s : Nat} (t : ArithTerm s) (v : Nat) :
    substTerm v (ArithTerm.var v) t = t := by
  induction t with
  | var n =>
    unfold substTerm
    split_ifs with h_vn
    · subst h_vn; rfl
    · rfl
  | const _ => rfl
  | henkin _ => rfl
  | zero => rfl
  | succ t' ih =>
    dsimp [substTerm]; rw [ih]
  | add t1 t2 ih1 ih2 =>
    dsimp [substTerm]; rw [ih1, ih2]
  | mul t1 t2 ih1 ih2 =>
    dsimp [substTerm]; rw [ih1, ih2]
  | exp t1 t2 ih1 ih2 =>
    dsimp [substTerm]; rw [ih1, ih2]
  | pred t' ih =>
    dsimp [substTerm]; rw [ih]

lemma substForm_self_var {s : Nat} (φ : ArithForm s) (v : Nat) :
    substForm v (ArithTerm.var v) φ = φ := by
  induction φ with
  | bottom => rfl
```

```
| eq t1 t2 =>
  dsimp [substForm]
  rw [substTerm_self_var t1 v, substTerm_self_var t2 v]
| impl f1 f2 ih1 ih2 =>
  dsimp [substForm]
  rw [ih1, ih2]
| neg f ih =>
  dsimp [substForm]
  rw [ih]
| forall_form n f ih =>
  dsimp [substForm]
  split_ifs with h_vn
  · rfl
  · rw [ih]
| provable_op heq t =>
     dsimp [substForm]
     rw [substTerm_self_var t v]
   | true_op heq t =>
     dsimp [substForm]
     rw [substTerm_self_var t v]
| list_nth_eq t1 t2 t3 =>
  dsimp [substForm]
  rw [substTerm_self_var t1 v, substTerm_self_var t2 v, substTerm_self_var t3 v]
```

```
def GammaWitnessesList : Nat → List (ArithForm 0)
 | 0 => []
 | m + 1 =>
  match getExistentialComponents (decodeExists (s := 0) m) with
  | some (v, ψ) =>
   let w := Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) + 1
   let freshEx := ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)))
   let c := ⟨Nat.max (maxHenkinList (GammaWitnessesList m)) (maxHenkinForm freshEx) + 1⟩
   ArithForm.impl freshEx (substForm v (ArithTerm.henkin c) ψ) :: GammaWitnessesList m
  | none => GammaWitnessesList m

def GammaSeq (Γ : Set (ArithForm 0)) : Nat → Set (ArithForm 0)
 | 0 => Γ
 | m + 1 =>
  match getExistentialComponents (decodeExists (s := 0) m) with
  | some (v, ψ) =>
   let w := Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) + 1
   let freshEx := ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)))
   let c := ⟨Nat.max (maxHenkinList (GammaWitnessesList m)) (maxHenkinForm freshEx) + 1⟩
   insert (ArithForm.impl freshEx (substForm v (ArithTerm.henkin c) ψ)) (GammaSeq Γ m)
  | none => GammaSeq Γ m
```

Límite de la secuencia de Henkin y monotonía

Se define el límite de la construcción de Henkin y se demuestran las propiedades de monotonía de la secuencia, que garantizan que la construcción es acumulativa y bien comportada.

Función `GammaLimit`: Límite de la secuencia de Henkin

Define el conjunto límite como la unión de todos los conjuntos de la secuencia:

$$ \text{GammaLimit}(\Gamma) := \bigcup_{n \in \mathbb{N}} \text{GammaSeq}(\Gamma, n) = \{\varphi \mid \exists n \in \mathbb{N},\ \varphi \in \text{GammaSeq}(\Gamma, n)\} $$

Intuición y necesidad

El límite de Henkin es la teoría resultante de añadir todos los axiomas de testigo para todas las fórmulas existenciales del lenguaje. Es el análogo sintáctico de la saturación existencial semántica: toda fórmula existencial que sea demostrable tendrá un testigo en el lenguaje expandido. Este conjunto será el punto de partida para la construcción de Lindenbaum.

Lema `GammaSeq_le_succ`: Crecimiento en un paso

Establece que cada paso de la secuencia extiende al paso anterior:

$$ \forall \Gamma \subseteq \text{ArithForm } 0,\ \forall n \in \mathbb{N},\ \text{GammaSeq}(\Gamma, n) \subseteq \text{GammaSeq}(\Gamma, n + 1) $$

La demostración es inmediata de la definición: cuando se procesa una fórmula existencial, el nuevo conjunto es `insert` del axioma en el conjunto anterior; cuando no, el conjunto permanece inalterado. En ambos casos, la inclusión se cumple.

Lema `GammaSeq_monotonic`: Monotonía global

Generaliza la propiedad anterior a cualquier par de índices ordenados:

$$ \forall \Gamma \subseteq \text{ArithForm } 0,\ \forall n, m \in \mathbb{N},\ n \leq m \implies \text{GammaSeq}(\Gamma, n) \subseteq \text{GammaSeq}(\Gamma, m) $$

La demostración es por inducción sobre la derivación de $n \leq m$, aplicando transitivamente `GammaSeq_le_succ` en cada paso intermedio.

Lema `GammaSeq_mono_step`: Versión simplificada

Proporciona una formulación directa para su uso en demostraciones:

$$ \forall \Gamma \subseteq \text{ArithForm } 0,\ \forall a \leq b,\ \text{GammaSeq}(\Gamma, a) \subseteq \text{GammaSeq}(\Gamma, b) $$

Significado

La monotonía de la secuencia de Henkin es una propiedad estructural fundamental. Garantiza que:

1. La construcción es acumulativa: ninguna fórmula se pierde al avanzar en la enumeración.

2. El límite está bien definido como supremo de una cadena creciente.

3. Cualquier fórmula que aparezca en algún paso finito pertenece a todos los pasos posteriores y, por tanto, al límite.

Esta propiedad es esencial para las demostraciones de consistencia y maximalidad del límite de Lindenbaum, ya que permite reducir razonamientos sobre el límite (potencialmente infinito) a razonamientos sobre pasos finitos de la secuencia, donde las propiedades pueden verificarse por inducción. La monotonía también se preserva en la construcción de Lindenbaum posterior (`LindenbaumSeq`), estableciendo un patrón uniforme para todas las construcciones de extensiones de teorías en el sistema SPA.

```
def GammaLimit (Γ : Set (ArithForm 0)) : Set (ArithForm 0) :=
 { φ | ∃ n, φ ∈ GammaSeq Γ n }

lemma GammaSeq_le_succ (Γ : Set (ArithForm 0)) (n : Nat) :
   GammaSeq Γ n ⊆ GammaSeq Γ (n + 1) := by
 intro x hx
 unfold GammaSeq
 split
```

```
  · exact Or.inr hx
  · exact hx

lemma GammaSeq_monotonic (Γ : Set (ArithForm 0)) (n m : Nat) (h : n ≤ m) :
    GammaSeq Γ n ⊆ GammaSeq Γ m := by
  induction h with
  | refl => exact Set.Subset.refl _
  | step _ ih => exact Set.Subset.trans ih (GammaSeq_le_succ Γ _)

lemma GammaSeq_mono_step (Γ : Set (ArithForm 0)) {a b : Nat} (h : a ≤ b) :
    GammaSeq Γ a ⊆ GammaSeq Γ b :=
  GammaSeq_monotonic Γ a b h
```

Consistencia y no ocurrencia de constantes de Henkin

Se define la noción de consistencia para teorías, y se introduce un predicado inductivo para garantizar que una constante de Henkin no aparece en una expresión sintáctica, junto con un lema que permite generar tal garantía por cota superior.

Definición `Consistent`: Teoría consistente

Define cuándo un conjunto de fórmulas Γ es consistente: no existe una derivación de $\bot$ a partir de Γ:

$$ \text{Consistent } \Gamma := \neg (\Gamma \vdash \bot) $$

Es decir, `Consistent Γ` es la proposición de que `SPA_Deduction_Internal s Γ ArithForm.bottom` implica `False`. Esta definición negativa captura la noción estándar de consistencia sintáctica.

Predicado `ConstantNotOccur`: No ocurrencia de constante de Henkin en un término

Define inductivamente cuándo una constante de Henkin c no aparece en un término t:

$$ \text{ConstantNotOccur } c\ t : \text{Prop} $$

Las reglas son análogas a las de `NotOccur`, pero referidas a constantes de Henkin:

- `var`, `const`, `zero`: Siempre verdadero (no contienen constantes de Henkin).
- `henkin c'`: Verdadero si $c' \neq c$.
- `succ`, `add`, `mul`, `exp`, `pred`: Se propaga recursivamente a los subtérminos.

Predicado `ConstantNotOccurForm`: No ocurrencia en fórmulas

Extiende el predicado a fórmulas:

$$ \text{ConstantNotOccurForm } c\ \varphi : \text{Prop} $$

con reglas análogas para cada constructor de `ArithForm`.

Definición `ConstantNotOccurSet`: No ocurrencia en un conjunto

Extiende a conjuntos de fórmulas:

$$ \text{ConstantNotOccurSet } c\ \Gamma := \forall \varphi \in \Gamma,\ \text{ConstantNotOccurForm } c\ \varphi $$

Lema `ConstantNotOccur_above_max_Henkin`: Frescura por cota superior para constantes

Establece que si el índice de una constante de Henkin c es estrictamente mayor que `maxHenkinTerm(t)`, entonces c no ocurre en t:

$$ \forall t \in \text{ArithTerm } s,\ \forall c \in \text{HenkinConst},\ c.\text{index} > \text{maxHenkinTerm}(t) \implies \text{ConstantNotOccur } c\ t $$

Estructura de la demostración

Por inducción sobre t:

- **`henkin c'`:** Si $c' = c$, entonces `c.index = c'.index ≤ maxHenkinTerm(t)`, contradiciendo la hipótesis. Por tanto $c' \neq c$ y se aplica el constructor `ConstantNotOccur.henkin`.

- **Casos binarios (`add`, `mul`, `exp`):** La hipótesis $c.\text{index} > \max(\text{maxHenkinTerm}(t_1), \text{maxHenkinTerm}(t_2))$ implica que supera cada cota individual, por lo que se aplican los lemas `lt_of_max_lt_left` y `lt_of_max_lt_right` para obtener las hipótesis de inducción.

- **Demás casos:** Se propagan directamente.

Significado

Estas definiciones y lemas son el análogo para constantes de Henkin de la maquinaria de frescura para variables. Así como `NotOccur` y `max_var_term` permiten generar variables frescas, `ConstantNotOccur` y `maxHenkinTerm` permiten generar constantes de Henkin frescas. La función `freshConst` utiliza precisamente `maxHenkinList + 1` para garantizar que la nueva constante no aparece en ninguna fórmula de la lista acumulada.

La preservación de la no ocurrencia de constantes de Henkin es crucial para la construcción de Henkin: cuando se añade un axioma de testigo con una nueva constante c, se debe garantizar que c no aparecía previamente en la teoría, para evitar inconsistencias como la paradoja del mentiroso. `ConstantNotOccur_above_max_Henkin` proporciona la garantía de que la constante generada por `freshConst` es efectivamente nueva. La definición de `Consistent` establece el criterio fundamental que la construcción de Henkin-Lindenbaum debe preservar en cada paso.

```
--
============================================================

-- CONSISTENCIA DE LA SECUENCIA DE HENKIN Y NO OCURRENCIA DE CONSTANTES

--
============================================================


def Consistent {s : Nat} (Γ : Set (ArithForm s)) : Prop :=

  SPA_Deduction_Internal s Γ ArithForm.bottom → False


inductive ConstantNotOccur (c : HenkinConst) {s : Nat} : ArithTerm s → Prop where

  | var (n : Nat) : ConstantNotOccur c (ArithTerm.var n)

  | const (c' : NatS s) : ConstantNotOccur c (ArithTerm.const c')

  | henkin (c' : HenkinConst) (h : c' ≠ c) : ConstantNotOccur c (ArithTerm.henkin c')

  | zero : ConstantNotOccur c ArithTerm.zero

  | succ (t : ArithTerm s) (h : ConstantNotOccur c t) : ConstantNotOccur c (ArithTerm.succ t)

  | add (t1 t2 : ArithTerm s) (h1 : ConstantNotOccur c t1) (h2 : ConstantNotOccur c t2) : ConstantNotOccur c (ArithTerm.add t1 t2)

  | mul (t1 t2 : ArithTerm s) (h1 : ConstantNotOccur c t1) (h2 : ConstantNotOccur c t2) : ConstantNotOccur c (ArithTerm.mul t1 t2)

  | exp (t1 t2 : ArithTerm s) (h1 : ConstantNotOccur c t1) (h2 : ConstantNotOccur c t2) : ConstantNotOccur c (ArithTerm.exp t1 t2)
```

```
 | pred (t : ArithTerm s) (h : ConstantNotOccur c t) : ConstantNotOccur c (ArithTerm.pred t)


inductive ConstantNotOccurForm (c : HenkinConst) {s : Nat} : ArithForm s → Prop where

 | bottom : ConstantNotOccurForm c ArithForm.bottom

 | eq (t1 t2 : ArithTerm s) (h1 : ConstantNotOccur c t1) (h2 : ConstantNotOccur c t2) : ConstantNotOccurForm c (ArithForm.eq t1 t2)

 | impl (f1 f2 : ArithForm s) (h1 : ConstantNotOccurForm c f1) (h2 : ConstantNotOccurForm c f2) : ConstantNotOccurForm c (ArithForm.impl f1 f2)

 | neg (f : ArithForm s) (h : ConstantNotOccurForm c f) : ConstantNotOccurForm c (ArithForm.neg f)

 | forall_form (n : Nat) (f : ArithForm s) (h : ConstantNotOccurForm c f) : ConstantNotOccurForm c (ArithForm.forall_form n f)

 | provable_op {s_lower : Nat} (h_eq : s = s_lower + 1) (t : ArithTerm s) (h : ConstantNotOccur c t) : ConstantNotOccurForm c (ArithForm.provable_op h_eq t)

 | true_op {s_lower : Nat} (h_eq : s = s_lower + 1) (t : ArithTerm s) (h : ConstantNotOccur c t) : ConstantNotOccurForm c (ArithForm.true_op h_eq t)

 | list_nth_eq (t1 t2 t3 : ArithTerm s) (h1 : ConstantNotOccur c t1) (h2 : ConstantNotOccur c t2) (h3 : ConstantNotOccur c t3) : ConstantNotOccurForm c (ArithForm.list_nth_eq t1 t2 t3)


def ConstantNotOccurSet {s : Nat} (c : HenkinConst) (Γ : Set (ArithForm s)) : Prop :=

 ∀ φ ∈ Γ, ConstantNotOccurForm c φ


lemma ConstantNotOccur_above_max_Henkin {s : Nat} (t : ArithTerm s) (c : HenkinConst) (h : c.index > maxHenkinTerm t) : ConstantNotOccur c t := by

 induction t with

 | var n => exact ConstantNotOccur.var n

 | const c' => exact ConstantNotOccur.const c'

 | henkin c' =>

  unfold maxHenkinTerm at h
```

```
  apply ConstantNotOccur.henkin
  intro h_eq
  subst h_eq
  omega
| zero => exact ConstantNotOccur.zero
| succ t' ih => unfold maxHenkinTerm at h; exact ConstantNotOccur.succ t' (ih h)
| add t1 t2 ih1 ih2 =>
  unfold maxHenkinTerm at h
  exact ConstantNotOccur.add t1 t2 (ih1 (lt_of_max_lt_left h)) (ih2 (lt_of_max_lt_right h))
| mul t1 t2 ih1 ih2 =>
  unfold maxHenkinTerm at h
  exact ConstantNotOccur.mul t1 t2 (ih1 (lt_of_max_lt_left h)) (ih2 (lt_of_max_lt_right h))
| exp t1 t2 ih1 ih2 =>
  unfold maxHenkinTerm at h
  exact ConstantNotOccur.exp t1 t2 (ih1 (lt_of_max_lt_left h)) (ih2 (lt_of_max_lt_right h))
| pred t' ih => unfold maxHenkinTerm at h; exact ConstantNotOccur.pred t' (ih h)
```

Lemas de preservación de cotas para constantes de Henkin bajo sustituciones

Se demuestran tres lemas que relacionan las cotas superiores de constantes de Henkin con las operaciones de sustitución. Estos resultados garantizan que las constantes frescas siguen siendo frescas después de realizar sustituciones.

Lema `ConstantNotOccurForm_of_gt_maxHenkinForm`: Frescura en fórmulas por cota superior

Extiende `ConstantNotOccur_above_max_Henkin` al nivel de fórmulas: si el índice de una constante de Henkin c supera `maxHenkinForm(φ)`, entonces c no ocurre en φ:

$$ \forall \varphi \in \text{ArithForm } } s,\ \forall c \in \text{HenkinConst},\ c.\text{index} > \text{maxHenkinForm}(\varphi) \implies \text{ConstantNotOccurForm } } c\ \varphi $$

La demostración es por inducción sobre φ, utilizando `ConstantNotOccur_above_max_Henkin` para los casos con términos y descomponiendo las cotas máximas en los casos compuestos mediante `lt_of_max_lt_left` y `lt_of_max_lt_right`.

Lema `maxHenkinTerm_substTerm`: Cota de constantes en un término tras sustitución

Establece que el máximo índice de constante de Henkin en un término sustituido está acotado por el máximo entre el índice en el término original y el índice en el término de sustitución:

$$ \text{maxHenkinTerm}(\text{substTerm } v\ u\ t) \le \max(\text{maxHenkinTerm}(t), \text{maxHenkinTerm}(u)) $$

La demostración es por inducción sobre t. Para `var n`, si hay sustitución se toma `maxHenkinTerm(u)`; si no, `maxHenkinTerm(t)`. Para los casos compuestos, se utiliza el lema `max_le_max_of_le` para combinar las cotas de los subtérminos.

Lema `maxHenkinForm_substForm`: Cota de constantes en una fórmula tras sustitución

Extiende el resultado anterior a fórmulas:

$$ \text{maxHenkinForm}(\text{substForm } v\ u\ \varphi) \le \max(\text{maxHenkinForm}(\varphi), \text{maxHenkinTerm}(u)) $$

La demostración es por inducción sobre φ, utilizando `maxHenkinTerm_substTerm` para los casos con términos y `max_le_max_of_le` para combinar cotas en los casos compuestos.

Significado

Estos lemas son técnicamente importantes para la construcción de Henkin porque garantizan que las constantes de Henkin frescas siguen siendo frescas después de realizar las sustituciones necesarias para construir los axiomas de testigo. En particular, cuando se construye el axioma $\text{freshEx} \to \psi[v := \text{henkin } c]$, se necesita asegurar que la nueva constante c no aparece en `freshEx` ni en ninguna otra fórmula previa. Los lemas de cota garantizan que `maxHenkinForm` de la fórmula sustituida no excede el máximo de las cotas originales, por lo que una constante con índice superior a todas ellas sigue siendo fresca.

La utilización recurrente de `max_le_max_of_le` en estas demostraciones refleja la estructura de retículo de las cotas: las operaciones sintácticas (sustitución, combinación de fórmulas) son monótonas respecto al orden de las cotas, lo que permite un manejo composicional de la frescura. Esta propiedad de monotonía es la que hace viable el enfoque de "frescura por cota superior" para garantizar que los nuevos símbolos no colisionan con los existentes.

```
lemma ConstantNotOccurForm_of_gt_maxHenkinForm {s : Nat} (φ : ArithForm s) (c : HenkinConst) (h : c.index > maxHenkinForm φ) : ConstantNotOccurForm c φ := by
  induction φ with
  | bottom => exact ConstantNotOccurForm.bottom
  | eq t1 t2 =>
    unfold maxHenkinForm at h
    exact ConstantNotOccurForm.eq t1 t2
      (ConstantNotOccur_above_max_Henkin t1 c (lt_of_max_lt_left h))
      (ConstantNotOccur_above_max_Henkin t2 c (lt_of_max_lt_right h))
  | impl f1 f2 ih1 ih2 =>
    unfold maxHenkinForm at h
    exact ConstantNotOccurForm.impl f1 f2 (ih1 (lt_of_max_lt_left h)) (ih2 (lt_of_max_lt_right h))
  | neg f ih =>
    unfold maxHenkinForm at h
    exact ConstantNotOccurForm.neg f (ih h)
  | forall_form n f ih =>
    unfold maxHenkinForm at h
    exact ConstantNotOccurForm.forall_form n f (ih h)
```

```
 | provable_op heq t =>
   unfold maxHenkinForm at h
   exact ConstantNotOccurForm.provable_op heq t (ConstantNotOccur_above_max_Henkin t c h)
 | true_op heq t =>
   unfold maxHenkinForm at h
   exact ConstantNotOccurForm.true_op heq t (ConstantNotOccur_above_max_Henkin t c h)
 | list_nth_eq t1 t2 t3 =>
   unfold maxHenkinForm at h
   exact ConstantNotOccurForm.list_nth_eq t1 t2 t3
    (ConstantNotOccur_above_max_Henkin t1 c (lt_of_max_lt_left h))
    (ConstantNotOccur_above_max_Henkin t2 c (lt_of_max_lt_left (lt_of_max_lt_right h)))
    (ConstantNotOccur_above_max_Henkin t3 c (lt_of_max_lt_right (lt_of_max_lt_right h)))

lemma maxHenkinTerm_substTerm {s : Nat} (t : ArithTerm s) (v : Nat) (u : ArithTerm s) :
   maxHenkinTerm (substTerm v u t) ≤ Nat.max (maxHenkinTerm t) (maxHenkinTerm u) := by
 induction t with
 | var n =>
   unfold substTerm
   split_ifs with _h_vn
   · exact Nat.le_max_right _ _
   · exact Nat.le_max_left _ _
 | const c' =>
   unfold substTerm; unfold maxHenkinTerm
   exact Nat.zero_le _
```

```
  | henkin c' =>
    unfold substTerm; unfold maxHenkinTerm
    exact Nat.le_max_left _ _
  | zero =>
    unfold substTerm; unfold maxHenkinTerm
    exact Nat.zero_le _
  | succ t' ih =>
    exact ih
  | add t1 t2 ih1 ih2 =>
    exact max_le_max_of_le ih1 ih2
  | mul t1 t2 ih1 ih2 =>
    exact max_le_max_of_le ih1 ih2
  | exp t1 t2 ih1 ih2 =>
    exact max_le_max_of_le ih1 ih2
  | pred t' ih =>
    exact ih

lemma maxHenkinForm_substForm {s : Nat} (φ : ArithForm s) (v : Nat) (u : ArithTerm
s) :
    maxHenkinForm (substForm v u φ) ≤ Nat.max (maxHenkinForm φ)
(maxHenkinTerm u) := by
  induction φ with
  | bottom =>
    unfold substForm; unfold maxHenkinForm
    exact Nat.zero_le _
  | eq t1 t2 =>
    exact max_le_max_of_le (maxHenkinTerm_substTerm t1 v u)
(maxHenkinTerm_substTerm t2 v u)
  | impl f1 f2 ih1 ih2 =>
```

```
  exact max_le_max_of_le ih1 ih2
| neg f ih =>
  exact ih
| forall_form n f ih =>
  unfold substForm
  split_ifs with _h_vn
  · exact Nat.le_max_left _ _
  · exact ih
| provable_op heq t =>
  exact maxHenkinTerm_substTerm t v u
| true_op heq t =>
  exact maxHenkinTerm_substTerm t v u
| list_nth_eq t1 t2 t3 =>
  exact max_le_max_of_le (maxHenkinTerm_substTerm t1 v u)
    (max_le_max_of_le (maxHenkinTerm_substTerm t2 v u) (maxHenkinTerm_substTerm t3 v u))
```

Lema `GammaSeq_maxHenkin_bound`: Cota uniforme para constantes en la secuencia de Henkin

Se demuestra un lema fundamental sobre la construcción de Henkin: si la teoría inicial Γ no contiene constantes de Henkin (todas las fórmulas tienen `maxHenkinForm = 0`), entonces en cualquier paso m de la secuencia, todas las fórmulas tienen su máximo índice de constante de Henkin acotado por `maxHenkinList` de la lista de testigos acumulada.

$$ \forall \Gamma \subseteq \text{ArithForm } 0,\ (\forall \varphi \in \Gamma,\ \text{maxHenkinForm } \varphi = 0) \implies $$

$$ \forall m \in \mathbb{N},\ \forall \varphi \in \text{GammaSeq } \Gamma\ m,\ \text{maxHenkinForm } \varphi \le \text{maxHenkinList}(\text{GammaWitnessesList } m) $$

Estructura de la demostración

La prueba procede por inducción sobre m:

Caso base ($m = 0$): `GammaSeq Γ 0 = Γ`. Por hipótesis, para toda $\varphi \in \Gamma$, `maxHenkinForm φ = 0`, y $0 \le \text{maxHenkinList}([])$. La cota se satisface trivialmente.

Paso inductivo ($m = k + 1$): Se analiza la definición de `GammaSeq`:

- Si `getExistentialComponents` devuelve `none`, el conjunto no cambia y la hipótesis de inducción se aplica directamente.

- Si devuelve `some (v, ψ)`, el nuevo conjunto es `insert` del axioma de testigo en el conjunto anterior. Se distinguen dos subcasos:

  - Si φ es el nuevo axioma, su `maxHenkinForm` está acotado por el máximo entre `maxHenkinForm(freshEx)` y `maxHenkinForm(ψ[v := henkin c])`. Por construcción, este máximo es exactamente `maxHenkinList(GammaWitnessesList (k+1))`, y se tiene la igualdad con la cota izquierda del máximo.

- Si φ pertenecía al conjunto anterior, por hipótesis de inducción su cota es $\le \text{maxHenkinList(GammaWitnessesList k)}$, y esta última está acotada por `maxHenkinList(GammaWitnessesList (k+1))` ya que la lista crece.

Significado

Este lema es crucial para garantizar que las constantes de Henkin generadas durante la construcción son efectivamente frescas. Establece que en cada paso m, todas las fórmulas acumuladas tienen sus constantes acotadas por el máximo de la lista de testigos. Por tanto, cuando se genera una nueva constante c con índice `maxHenkinList(GammaWitnessesList m) + 1`, esta constante no aparece en ninguna fórmula del conjunto `GammaSeq Γ m`.

La hipótesis de que Γ inicial no contiene constantes de Henkin es natural: la teoría original está formulada en el lenguaje sin constantes de Henkin (solo contiene variables, numerales y los constructores estándar). Todas las constantes de Henkin son introducidas por la propia construcción. El lema garantiza que este proceso de introducción es controlado y no produce colisiones. Esta propiedad de "frescura controlada" es la que permite demostrar que la extensión de Henkin preserva la consistencia: si la teoría original era consistente, añadir axiomas de testigo con constantes nuevas no puede generar una contradicción, ya que las nuevas constantes no aparecen en las fórmulas previas y por tanto no pueden participar en ninguna derivación de $\bot$.

```
lemma GammaSeq_maxHenkin_bound (Γ : Set (ArithForm 0))
  (h_initial : ∀ φ ∈ Γ, maxHenkinForm φ = 0) (m : Nat) :
  ∀ φ ∈ GammaSeq Γ m, maxHenkinForm φ ≤ maxHenkinList (GammaWitnessesList m) := by
 induction m with
 | zero =>
```

```
  intro φ h_mem
  unfold GammaSeq at h_mem
  have h_zero : maxHenkinForm φ = 0 := h_initial φ h_mem
  rw [h_zero]
  exact Nat.zero_le _
| succ k ih =>
  intro φ h_mem
  unfold GammaSeq at h_mem
  unfold GammaWitnessesList
  split at h_mem
  · rename_i v ψ h_dec
    rcases h_mem with rfl | h_mem_old
    · exact Nat.le_max_left _ _
    · have h_old_bound := ih φ h_mem_old
      exact Nat.le_trans h_old_bound (Nat.le_max_right _ _)
  · exact ih φ h_mem
```

Sustitución de constantes de Henkin y lemas de frescura

Se define la operación de sustitución de constantes de Henkin por términos, que es el análogo para constantes de la sustitución de variables. Esta operación es fundamental para la demostración de que la extensión de Henkin es conservativa.

Función `substHenkinTerm`: Sustitución de constante de Henkin en un término

Reemplaza todas las ocurrencias de una constante de Henkin c por un término t dentro de un término:

$$ \text{substHenkinTerm}(c, t, \cdot) : \text{ArithTerm } s \to \text{ArithTerm } s $$

La definición es recursiva estructural:

- `var`, `const`, `zero`: Inalterados.

- `henkin c'`: Si $c' = c$, se reemplaza por t; en caso contrario, se preserva.

- `succ`, `add`, `mul`, `exp`, `pred`: Se propaga recursivamente a los subtérminos.

Función `substHenkinForm`: Sustitución de constante de Henkin en una fórmula

Extiende la operación a fórmulas, propagándose a través de todos los constructores de `ArithForm`:

$$ \text{substHenkinForm}(c, t, \cdot) : \text{ArithForm } s \to \text{ArithForm } s $$

Lema `substHenkinTerm_fresh`: Invarianza bajo sustitución fresca

Establece que si una constante de Henkin c no ocurre en un término t, entonces sustituir c por cualquier término u deja t inalterado:

$$ \forall t, u \in \text{ArithTerm } s,\ \forall c \in \text{HenkinConst},\ \text{ConstantNotOccur } c\ t \implies \text{substHenkinTerm } c\ u\ t = t $$

La demostración es por inducción sobre t, utilizando inversión sobre `ConstantNotOccur c t` para obtener las condiciones sobre los subtérminos. El caso `henkin c'` es el único no trivial: la hipótesis `ConstantNotOccur c (henkin c')` fuerza $c' \neq c$, por lo que la sustitución no afecta.

Lema `substHenkinForm_fresh`: Invarianza en fórmulas bajo sustitución fresca

Extiende el resultado anterior a fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall c \in \text{HenkinConst},\ \forall u \in \text{ArithTerm } s,\ \text{ConstantNotOccurForm } c\ \varphi \implies \text{substHenkinForm } c\ u\ \varphi = \varphi $$

La demostración es por inducción sobre φ, utilizando `substHenkinTerm_fresh` para los casos con términos y las hipótesis de inducción para los casos recursivos.

Significado

Estas definiciones y lemas son la base técnica para uno de los pasos más delicados de la prueba de completitud: la eliminación de constantes de Henkin. La idea es que cualquier demostración que utilice constantes de Henkin puede ser transformada en una demostración que no las utiliza, reemplazando cada constante por una variable fresca adecuada y aplicando generalización universal. Los lemas `substHenkinTerm_fresh` y `substHenkinForm_fresh` garantizan que las fórmulas que no contienen la constante eliminada no se ven afectadas por la sustitución, lo cual es esencial para preservar la estructura de la demostración original durante el proceso de eliminación. Esta técnica, conocida como "eliminación de constantes de Henkin" o "Henkinization lemma", es el puente entre la teoría expandida (con testigos) y la teoría original (sin ellos), y constituye el núcleo de la demostración de completitud.

```
--
============================================================

-- DEFINICIÓN DE SUSTITUCIÓN DE CONSTANTES DE HENKIN (FASE 8)

--
============================================================


def substHenkinTerm {s : Nat} (c : HenkinConst) (t : ArithTerm s) : ArithTerm s →
ArithTerm s

  | ArithTerm.var n => ArithTerm.var n

  | ArithTerm.const c' => ArithTerm.const c'

  | ArithTerm.henkin c' => if c' = c then t else ArithTerm.henkin c'

  | ArithTerm.zero => ArithTerm.zero

  | ArithTerm.succ t' => ArithTerm.succ (substHenkinTerm c t t')
```

```
 | ArithTerm.add t1 t2 => ArithTerm.add (substHenkinTerm c t t1) (substHenkinTerm c t t2)

 | ArithTerm.mul t1 t2 => ArithTerm.mul (substHenkinTerm c t t1) (substHenkinTerm c t t2)

 | ArithTerm.exp t1 t2 => ArithTerm.exp (substHenkinTerm c t t1) (substHenkinTerm c t t2)

 | ArithTerm.pred t' => ArithTerm.pred (substHenkinTerm c t t')


def substHenkinForm {s : Nat} (c : HenkinConst) (t : ArithTerm s) : ArithForm s → ArithForm s

 | ArithForm.bottom => ArithForm.bottom

 | ArithForm.eq t1 t2 => ArithForm.eq (substHenkinTerm c t t1) (substHenkinTerm c t t2)

 | ArithForm.impl f1 f2 => ArithForm.impl (substHenkinForm c t f1) (substHenkinForm c t f2)

 | ArithForm.neg f => ArithForm.neg (substHenkinForm c t f)

 | ArithForm.forall_form n f => ArithForm.forall_form n (substHenkinForm c t f)

 | ArithForm.provable_op h t' => ArithForm.provable_op h (substHenkinTerm c t t')

 | ArithForm.true_op h t' => ArithForm.true_op h (substHenkinTerm c t t')

 | ArithForm.list_nth_eq t1 t2 t3 => ArithForm.list_nth_eq (substHenkinTerm c t t1) (substHenkinTerm c t t2) (substHenkinTerm c t t3)


--
================================================================

-- LEMAS DE FRESCURA DE VARIABLES Y CONSTANTES

--
================================================================


lemma substHenkinTerm_fresh {s : Nat} (t : ArithTerm s) (c : HenkinConst) (u : ArithTerm s) (h : ConstantNotOccur c t) :

   substHenkinTerm c u t = t := by

 induction t with
```

```
  | var _ => rfl
  | const _ => rfl
  | henkin c' =>
    unfold substHenkinTerm
    cases h with | henkin _ h_ne =>
    split_ifs with h_eq
    · subst h_eq; contradiction
    · rfl
  | zero => rfl
  | succ _ ih => cases h with | succ _ h_sub => unfold substHenkinTerm; rw [ih h_sub]
  | add _ _ ih1 ih2 => cases h with | add _ _ h1 h2 => unfold substHenkinTerm; rw [ih1 h1, ih2 h2]
  | mul _ _ ih1 ih2 => cases h with | mul _ _ h1 h2 => unfold substHenkinTerm; rw [ih1 h1, ih2 h2]
  | exp _ _ ih1 ih2 => cases h with | exp _ _ h1 h2 => unfold substHenkinTerm; rw [ih1 h1, ih2 h2]
  | pred _ ih => cases h with | pred _ h_sub => unfold substHenkinTerm; rw [ih h_sub]


lemma substHenkinForm_fresh {s : Nat} (φ : ArithForm s) (c : HenkinConst) (u : ArithTerm s) (h : ConstantNotOccurForm c φ) :
    substHenkinForm c u φ = φ := by
  induction φ with
  | bottom => rfl
  | eq t1 t2 =>
    cases h with | eq _ _ h1 h2 =>
    dsimp [substHenkinForm]
    rw [substHenkinTerm_fresh t1 c u h1, substHenkinTerm_fresh t2 c u h2]
  | impl f1 f2 ih1 ih2 =>
    cases h with | impl _ _ h1 h2 =>
    dsimp [substHenkinForm]
```

```
    rw [ih1 h1, ih2 h2]
  | neg f ih =>
    cases h with | neg _ h' =>
    dsimp [substHenkinForm]
    rw [ih h']
  | forall_form n f ih =>
    cases h with | forall_form _ _ h' =>
    dsimp [substHenkinForm]
    rw [ih h']
  | provable_op heq t =>
    cases h with | provable_op _ _ h' =>
    dsimp [substHenkinForm]
    rw [substHenkinTerm_fresh t c u h']
  | true_op heq t =>
    cases h with | true_op _ _ h' =>
    dsimp [substHenkinForm]
    rw [substHenkinTerm_fresh t c u h']
  | list_nth_eq t1 t2 t3 =>
    cases h with | list_nth_eq _ _ _ h1 h2 h3 =>
    dsimp [substHenkinForm]
    rw [substHenkinTerm_fresh t1 c u h1, substHenkinTerm_fresh t2 c u h2,
substHenkinTerm_fresh t3 c u h3]
```

Lemas de preservación de no ocurrencia bajo sustitución de constantes de Henkin

Se demuestran dos lemas que establecen que la propiedad de no ocurrencia de variables se preserva bajo la operación de sustitución de constantes de Henkin. Estos resultados son análogos a los lemas `NotOccur_subst_other` y `NotOccurForm_subst_other`, pero para el nuevo tipo de sustitución.

Lema `NotOccur_substHenkinTerm`: Preservación de no ocurrencia en términos

Establece que si una variable v no ocurre en un término t ni en un término u, entonces v tampoco ocurre en el resultado de sustituir la constante de Henkin c por u en t:

$$ \forall t, u \in \text{ArithTerm } s,\ \forall c \in \text{HenkinConst},\ \forall v \in \mathbb{N}, $$

$$ \text{NotOccur } v\ t \land \text{NotOccur } v\ u \implies \text{NotOccur } v\ (\text{substHenkinTerm } c\ u\ t) $$

Estructura de la demostración

Por inducción sobre t:

- **`var n`, `const`, `zero`:** La sustitución no modifica el término, por lo que `NotOccur v t` se preserva directamente.

- **`henkin c'`:** Si $c' = c$, se reemplaza por u, y la hipótesis `NotOccur v u` garantiza el resultado. Si $c' \neq c$, se preserva como `henkin c'`, que no contiene variables.

- **Casos recursivos:** Se invierte la hipótesis `NotOccur v t` para obtener las condiciones sobre los subtérminos, y se aplican las hipótesis de inducción.

Lema `NotOccurForm_substHenkinForm`: Preservación de no ocurrencia en fórmulas

Extiende el resultado anterior al nivel de fórmulas: si una variable v no ocurre en una fórmula φ ni en un término u, entonces v no ocurre en φ tras sustituir c por u:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall c \in \text{HenkinConst},\ \forall u \in \text{ArithTerm } s,\ \forall v \in \mathbb{N}, $$

$$ \text{NotOccurForm } v\ \varphi \land \text{NotOccur } v\ u \implies \text{NotOccurForm } v\ (\text{substHenkinForm } c\ u\ \varphi) $$

La demostración es por inducción sobre φ, utilizando `NotOccur_substHenkinTerm` para los casos que involucran términos y las hipótesis de inducción para los casos recursivos. En `forall_form n f`, la condición $v \neq n$ de `NotOccurForm` se preserva porque la sustitución no modifica las variables ligadas.

Significado

Estos lemas garantizan que la operación de eliminación de constantes de Henkin es "inocua" respecto a las variables del lenguaje original: si una variable no aparecía en una fórmula antes de eliminar las constantes de Henkin, tampoco aparecerá después. Esta propiedad es esencial para la demostración de que la extensión de Henkin es conservativa: al reemplazar las constantes de Henkin por variables frescas, no se introducen nuevas ocurrencias de variables que pudieran violar las condiciones de las reglas de inferencia (como la generalización universal).

La estructura de las demostraciones refleja un principio general: las operaciones de sustitución (ya sea de variables o de constantes) son "funciones" sobre la sintaxis que preservan las propiedades de no ocurrencia siempre que el término de sustitución también las satisfaga. Este principio de composicionalidad de la frescura es ubicuo en el desarrollo y es lo que permite manipular fórmulas complejas manteniendo el control sobre qué variables y constantes aparecen en ellas.

```
lemma NotOccur_substHenkinTerm {s : Nat} (t : ArithTerm s) (c : HenkinConst) (u :
ArithTerm s) (v : Nat)
   (h_t : NotOccur v t) (h_u : NotOccur v u) :
   NotOccur v (substHenkinTerm c u t) := by
 induction t with
 | var _ => unfold substHenkinTerm; exact h_t
```

```
  | const _ => exact NotOccur.const _
  | henkin _ =>
    unfold substHenkinTerm
    split_ifs with _h_eq
    · exact h_u
    · exact NotOccur.henkin _
  | zero => exact NotOccur.zero
  | succ _ ih =>
    cases h_t with | succ _ h_sub =>
    unfold substHenkinTerm; exact NotOccur.succ _ (ih h_sub)
  | add _ _ ih1 ih2 =>
    cases h_t with | add _ _ h1 h2 =>
    unfold substHenkinTerm; exact NotOccur.add _ _ (ih1 h1) (ih2 h2)
  | mul _ _ ih1 ih2 =>
    cases h_t with | mul _ _ h1 h2 =>
    unfold substHenkinTerm; exact NotOccur.mul _ _ (ih1 h1) (ih2 h2)
  | exp _ _ ih1 ih2 =>
    cases h_t with | exp _ _ h1 h2 =>
    unfold substHenkinTerm; exact NotOccur.exp _ _ (ih1 h1) (ih2 h2)
  | pred _ ih =>
    cases h_t with | pred _ h_sub =>
    unfold substHenkinTerm; exact NotOccur.pred _ (ih h_sub)

lemma NotOccurForm_substHenkinForm {s : Nat} (φ : ArithForm s) (c : HenkinConst)
(u : ArithTerm s) (v : Nat)
    (h_φ : NotOccurForm v φ) (h_u : NotOccur v u) :
    NotOccurForm v (substHenkinForm c u φ) := by
  induction φ with
```

```
  | bottom => trivial
  | eq t1 t2 =>
    unfold NotOccurForm at h_φ ⊢
    unfold substHenkinForm
    exact ⟨NotOccur_substHenkinTerm t1 c u v h_φ.1 h_u, NotOccur_substHenkinTerm
t2 c u v h_φ.2 h_u⟩
  | impl f1 f2 ih1 ih2 =>
    unfold NotOccurForm at h_φ ⊢
    unfold substHenkinForm
    exact ⟨ih1 h_φ.1, ih2 h_φ.2⟩
  | neg f ih =>
    unfold NotOccurForm at h_φ ⊢
    unfold substHenkinForm
    exact ih h_φ
  | forall_form n f ih =>
    unfold NotOccurForm at h_φ ⊢
    unfold substHenkinForm
    exact ⟨h_φ.1, ih h_φ.2⟩
  | provable_op heq t =>
    unfold NotOccurForm at h_φ ⊢
    unfold substHenkinForm
    exact NotOccur_substHenkinTerm t c u v h_φ h_u
  | true_op heq t =>
    unfold NotOccurForm at h_φ ⊢
    unfold substHenkinForm
    exact NotOccur_substHenkinTerm t c u v h_φ h_u
  | list_nth_eq t1 t2 t3 =>
    unfold NotOccurForm at h_φ ⊢
```

```
  unfold substHenkinForm

  exact ⟨NotOccur_substHenkinTerm t1 c u v h_φ.1 h_u, NotOccur_substHenkinTerm t2 c u v h_φ.2.1 h_u, NotOccur_substHenkinTerm t3 c u v h_φ.2.2 h_u⟩
```

Lema `NotFreeForm_substHenkinForm` y `NotFreeInSet_substHenkin`

Se completa la infraestructura de preservación de frescura bajo sustitución de constantes de Henkin, extendiendo los resultados al predicado `NotFreeForm` (no libertad) y a conjuntos de fórmulas.

Lema `NotFreeForm_substHenkinForm`: Preservación de no libertad bajo sustitución de constantes

Establece que si una variable v no es libre en una fórmula φ y no ocurre en un término u, entonces v tampoco es libre en φ tras sustituir la constante de Henkin c por u:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall c \in \text{HenkinConst},\ \forall u \in \text{ArithTerm } s,\ \forall v \in \mathbb{N}, $$

$$ \text{NotFreeForm } v\ \varphi \land \text{NotOccur } v\ u \implies \text{NotFreeForm } v\ (\text{substHenkinForm } c\ u\ \varphi) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos que involucran términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`):** `NotFreeForm` se reduce a `NotOccur` sobre los subtérminos, y se aplica `NotOccur_substHenkinTerm` a cada uno.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción a las subfórmulas.

- **`forall_form n f`:** Se distingue si $v = n$ o no:

  - Si $v = n$, la variable está ligada y `NotFreeForm` es trivial, tanto antes como después de la sustitución (el cuantificador no se modifica).

  - Si $v \neq n$, `NotFreeForm` en la fórmula original implica que v no es libre en f, y la hipótesis de inducción garantiza que tampoco lo es en `substHenkinForm c u f`. La sustitución no afecta al cuantificador, por lo que la propiedad se preserva.

Lema `NotFreeInSet_substHenkin`: Preservación en conjuntos de fórmulas

Extiende el resultado a conjuntos: si una variable v no es libre en ninguna fórmula de un conjunto Γ, entonces tampoco es libre en las fórmulas resultantes de sustituir c por u en Γ:

$$ \forall v \in \mathbb{N},\ \forall \Gamma \subseteq \text{ArithForm } s,\ \forall c \in \text{HenkinConst},\ \forall u \in \text{ArithTerm } s, $$

$$ \text{NotFreeInSet } v\ \Gamma \land \text{NotOccur } v\ u \implies \text{NotFreeInSet } v\ \{\text{substHenkinForm } c\ u\ \psi \mid \psi \in \Gamma\} $$

La demostración es directa: para cada fórmula ψ' en el conjunto imagen, existe $\psi \in \Gamma$ tal que $\psi' = \text{substHenkinForm } c\ u\ \psi$, y se aplica `NotFreeForm_substHenkinForm` usando que v no es libre en ψ (por `h_vΓ`) y no ocurre en u.

Significado

Estos lemas son cruciales para la demostración de que la eliminación de constantes de Henkin preserva la estructura deductiva. En particular, cuando se aplica la regla de generalización universal (`forall_intro`), se requiere que la variable generalizada no sea libre en el conjunto de hipótesis. Si se han eliminado constantes de Henkin reemplazándolas por variables frescas, es necesario garantizar que estas nuevas variables no aparecen libres en las hipótesis. `NotFreeInSet_substHenkin` garantiza precisamente esto: si las variables utilizadas para reemplazar las constantes de Henkin son frescas (no ocurren en las hipótesis originales), entonces siguen sin ser libres en las hipótesis transformadas.

Este control sobre las variables libres es lo que permite cerrar el círculo de la prueba de completitud: partiendo de una demostración en la teoría expandida (con constantes de Henkin), se reemplazan todas las constantes por variables frescas, se verifica que las reglas de inferencia siguen siendo aplicables (porque las condiciones de frescura se preservan), y finalmente se generaliza para obtener una demostración en la teoría original. Los lemas de preservación de frescura garantizan que este proceso de traducción es sintácticamente correcto.

```
lemma NotFreeForm_substHenkinForm {s : Nat} (φ : ArithForm s) (c : HenkinConst)
(u : ArithTerm s) (v : Nat)
  (h_φ : NotFreeForm v φ) (h_u : NotOccur v u) :
  NotFreeForm v (substHenkinForm c u φ) := by
 induction φ with
 | bottom => trivial
 | eq t1 t2 =>
  unfold NotFreeForm at h_φ ⊢
  unfold substHenkinForm
  exact ⟨NotOccur_substHenkinTerm t1 c u v h_φ.1 h_u, NotOccur_substHenkinTerm
t2 c u v h_φ.2 h_u⟩
 | impl f1 f2 ih1 ih2 =>
  unfold NotFreeForm at h_φ ⊢
  unfold substHenkinForm
  exact ⟨ih1 h_φ.1, ih2 h_φ.2⟩
 | neg f ih =>
  unfold NotFreeForm at h_φ ⊢
  unfold substHenkinForm
  exact ih h_φ
 | forall_form n f ih =>
  dsimp [NotFreeForm, substHenkinForm] at h_φ ⊢
  by_cases h_vn : v = n
  · rw [if_pos h_vn] at h_φ ⊢
    trivial
```

```
    · rw [if_neg h_vn] at h_φ ⊢
      exact ih h_φ
  | provable_op heq t =>
    unfold NotFreeForm at h_φ ⊢
    unfold substHenkinForm
    exact NotOccur_substHenkinTerm t c u v h_φ h_u
  | true_op heq t =>
    unfold NotFreeForm at h_φ ⊢
    unfold substHenkinForm
    exact NotOccur_substHenkinTerm t c u v h_φ h_u
  | list_nth_eq t1 t2 t3 =>
    unfold NotFreeForm at h_φ ⊢
    unfold substHenkinForm
    exact ⟨NotOccur_substHenkinTerm t1 c u v h_φ.1 h_u, NotOccur_substHenkinTerm
t2 c u v h_φ.2.1 h_u, NotOccur_substHenkinTerm t3 c u v h_φ.2.2 h_u⟩


lemma NotFreeInSet_substHenkin {s : Nat} (v : Nat) (Γ : Set (ArithForm s)) (c :
HenkinConst) (u : ArithTerm s)
    (h_vΓ : NotFreeInSet v Γ) (h_u : NotOccur v u) :
    NotFreeInSet v { substHenkinForm c u ψ | ψ ∈ Γ } := by
  intro ψ' h_mem
  rcases h_mem with ⟨ψ, h_ψ, rfl⟩
  exact NotFreeForm_substHenkinForm ψ c u v (h_vΓ ψ h_ψ) h_u
```

Lemas de conmutación entre sustitución de constantes de Henkin y sustitución de variables

Se demuestran dos lemas fundamentales que establecen que la sustitución de constantes de Henkin conmuta con la sustitución ordinaria de variables, siempre que las variables involucradas sean distintas. Estos resultados son técnicamente esenciales para la eliminación de constantes de Henkin en fórmulas que también involucran sustituciones de variables.

Lema `substHenkinTerm_substTerm_var`: Conmutación en términos

Establece que sustituir primero una variable v por t y luego una constante de Henkin c por $\text{var } w$ es equivalente a hacerlo en orden inverso, siempre que $v \neq w$:

$$ \forall t' \in \text{ArithTerm } s,\ \forall c \in \text{HenkinConst},\ \forall w, v \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ v \neq w \implies $$

$$ \text{substHenkinTerm } c\ (\text{var } w)\ (\text{substTerm } v\ t\ t') = \text{substTerm } v\ (\text{substHenkinTerm } c\ (\text{var } w)\ t)\ (\text{substHenkinTerm } c\ (\text{var } w)\ t') $$

Estructura de la demostración

Por inducción sobre t':

- **`var n`:** Ambos lados se reducen a un análisis de casos sobre si $n = v$ y si el resultado es una variable o el término t. La condición $v \neq w$ garantiza que la sustitución de Henkin no interfiere con la sustitución de variable.

- **`henkin c'`:** Si $c' = c$, la sustitución de Henkin reemplaza la constante por $\text{var } w$. La sustitución de variable sobre $\text{var } w$ (si $v = w$) no ocurre porque $v \neq w$, garantizando la conmutación.

- **Casos recursivos:** Se aplican las hipótesis de inducción a los subtérminos.

Lema `substHenkinForm_substForm_var`: Conmutación en fórmulas

Extiende el resultado al nivel de fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall c \in \text{HenkinConst},\ \forall w, v \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ v \neq w \implies $$

$$ \text{substHenkinForm } c\ (\text{var } w)\ (\text{substForm } v\ t\ \varphi) = \text{substForm } v\ (\text{substHenkinTerm } c\ (\text{var } w)\ t)\ (\text{substHenkinForm } c\ (\text{var } w)\ \varphi) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos con términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`):** Se aplica `substHenkinTerm_substTerm_var` a cada subtérmino.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Se distingue si $v = n$:

- Si $v = n$, la sustitución de variable no penetra el cuantificador, y la sustitución de Henkin tampoco lo modifica. Ambos lados coinciden.

- Si $v \neq n$, la sustitución de variable atraviesa el cuantificador, y la hipótesis de inducción se aplica al cuerpo f.

Significado

Estos lemas de conmutación son la pieza técnica central para la eliminación de constantes de Henkin en derivaciones. Durante la prueba de completitud, será necesario transformar una derivación que usa constantes de Henkin en otra que no las usa. Esta transformación implica reemplazar cada constante de Henkin c por una variable fresca w, lo cual requiere reescribir todas las fórmulas de la derivación usando `substHenkinForm`.

Sin embargo, la derivación original puede contener aplicaciones de `forall_elim` que realizan sustituciones de variables. Los lemas de conmutación garantizan que el orden en que se realizan estas dos operaciones (eliminación de constantes y sustitución de variables) es irrelevante, siempre que las variables sean distintas. Esto permite "empujar" la sustitución de Henkin a través de toda la derivación de manera uniforme, preservando la estructura de la prueba.

La condición $v \neq w$ es crucial: si la variable de sustitución v coincidiera con la variable fresca w utilizada para reemplazar la constante, se produciría una captura indeseada. En la práctica, w se elige siempre como una variable fresca que no aparece en la derivación original, por lo que esta condición se satisface automáticamente.

```
--
================================================================

-- CONMUTACIÓN CON LA SUSTITUCIÓN ORDINARIA DE VARIABLES

--
================================================================


lemma substHenkinTerm_substTerm_var {s : Nat} (t' : ArithTerm s) (c : HenkinConst)
(w : Nat) (v : Nat) (t : ArithTerm s) (h_vw : v ≠ w) :

  substHenkinTerm c (ArithTerm.var w) (substTerm v t t') = substTerm v
(substHenkinTerm c (ArithTerm.var w) t) (substHenkinTerm c (ArithTerm.var w) t')
:= by

 induction t' with

 | var n =>

  dsimp [substTerm, substHenkinTerm]

  split_ifs <;> rfl

 | const c' => rfl

 | henkin c' =>

  dsimp [substTerm, substHenkinTerm]

  split_ifs with h_eq

  · subst h_eq

   dsimp [substTerm, substHenkinTerm]

   split_ifs with h_vw'

   · subst h_vw'; contradiction

   · rfl

  · rfl

 | zero => rfl

 | succ t'' ih =>
```

```
  dsimp [substTerm, substHenkinTerm]
  rw [ih]
| add t1 t2 ih1 ih2 =>
  dsimp [substTerm, substHenkinTerm]
  rw [ih1, ih2]
| mul t1 t2 ih1 ih2 =>
  dsimp [substTerm, substHenkinTerm]
  rw [ih1, ih2]
| exp t1 t2 ih1 ih2 =>
  dsimp [substTerm, substHenkinTerm]
  rw [ih1, ih2]
| pred t'' ih =>
  dsimp [substTerm, substHenkinTerm]
  rw [ih]

lemma substHenkinForm_substForm_var {s : Nat} (φ : ArithForm s) (c : HenkinConst)
(w : Nat) (v : Nat) (t : ArithTerm s) (h_vw : v ≠ w) :
  substHenkinForm c (ArithTerm.var w) (substForm v t φ) = substForm v
(substHenkinTerm c (ArithTerm.var w) t) (substHenkinForm c (ArithTerm.var w) φ)
:= by
 induction φ generalizing t with
 | bottom => rfl
 | eq t1 t2 =>
  dsimp [substForm, substHenkinForm]
  rw [substHenkinTerm_substTerm_var t1 c w v t h_vw,
substHenkinTerm_substTerm_var t2 c w v t h_vw]
 | impl f1 f2 ih1 ih2 =>
  dsimp [substForm, substHenkinForm]
  rw [ih1 t, ih2 t]
 | neg f ih =>
```

```
  dsimp [substForm, substHenkinForm]
  rw [ih t]
| forall_form n f ih =>
  dsimp [substHenkinForm, substForm]
  split_ifs with h_vn
  · rfl
  · dsimp [substHenkinForm, substForm]
    rw [ih t]
| provable_op heq t' =>
  dsimp [substForm, substHenkinForm]
  rw [substHenkinTerm_substTerm_var t' c w v t h_vw]
| true_op heq t' =>
  dsimp [substForm, substHenkinForm]
  rw [substHenkinTerm_substTerm_var t' c w v t h_vw]
| list_nth_eq t1 t2 t3 =>
  dsimp [substForm, substHenkinForm]
  rw [substHenkinTerm_substTerm_var t1 c w v t h_vw, substHenkinTerm_substTerm_var t2 c w v t h_vw, substHenkinTerm_substTerm_var t3 c w v t h_vw]
```

Lemas de no ocurrencia y preservación de libertad bajo sustitución de Henkin

Se demuestran dos lemas que refinan las propiedades de la sustitución de constantes de Henkin: el primero garantiza que las variables originales no ocurren en el resultado si no ocurrían en el término original; el segundo establece que la condición de "ser libre para" se preserva bajo la sustitución de Henkin.

Lema `NotOccur_substHenkinTerm_var`: Preservación de no ocurrencia con variable fresca

Establece que si una variable n no ocurre en un término t y $n \neq w$, entonces n tampoco ocurre en t tras sustituir la constante c por $\text{var } w$:

$$ \forall t \in \text{ArithTerm } s,\ \forall c \in \text{HenkinConst},\ \forall w, n \in \mathbb{N},\ \text{NotOccur } n\ t \land n \neq w \implies \text{NotOccur } n\ (\text{substHenkinTerm } c\ (\text{var } w)\ t) $$

Estructura de la demostración

Por inducción sobre t:

- **`var n'`, `const`, `zero`:** La sustitución no modifica el término, por lo que `NotOccur n t` se preserva.

- **`henkin c'`:** Si $c' = c$, la constante se reemplaza por $\text{var } w$. Dado que $n \neq w$, `NotOccur n (var w)` se cumple. Si $c' \neq c$, se preserva como `henkin c'`, que no contiene variables.

- **Casos recursivos:** Se invierte `NotOccur n t` y se aplican las hipótesis de inducción.

Intuición

La sustitución de una constante de Henkin por una variable w solo puede introducir la variable w en el término. Si n es distinta de w y no aparecía originalmente, sigue sin aparecer tras la sustitución.

Lema `IsFreeFor_substHenkin_var`: Preservación de la libertad bajo sustitución de Henkin

Establece que si un término t es libre para una variable v en una fórmula φ, entonces el término transformado (sustituyendo c por $\text{var } w$) es libre para v en la fórmula transformada, siempre que w no esté ligada en φ y $v \neq w$:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \forall c \in \text{HenkinConst},\ \forall w \in \mathbb{N}, $$

$$ v \neq w \land \text{NoBoundVar } w\ \varphi \land \text{IsFreeFor } v\ t\ \varphi \implies $$

$$ \text{IsFreeFor } v\ (\text{substHenkinTerm } c\ (\text{var } w)\ t)\ (\text{substHenkinForm } c\ (\text{var } w)\ \varphi) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos base (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`):** `IsFreeFor` es trivialmente verdadero para estos constructores.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción a las subfórmulas.

- **`forall_form n f`:** El caso más delicado. Si $v = n$, la sustitución no penetra y la condición es trivial. Si $v \neq n$, la hipótesis `IsFreeFor v t (forall_form n f)` proporciona:

  - `NotOccur n t`: n no ocurre en t.

  - `IsFreeFor v t f`: t es libre para v en f.

  Se debe demostrar:

  - `NotOccur n (substHenkinTerm c (var w) t)`: Por `NotOccur_substHenkinTerm_var`, usando que $n \neq w$ (provisto por `NoBoundVar w (forall_form n f)`).

  - `IsFreeFor v (substHenkinTerm c (var w) t) (substHenkinForm c (var w) f)`: Por hipótesis de inducción sobre f.

Significado

`IsFreeFor_substHenkin_var` es uno de los lemas más técnicamente sofisticados de esta sección. Garantiza que la eliminación de constantes de Henkin no interfiere con la aplicabilidad de la regla `forall_elim`. En la demostración de completitud, cuando se transforma una derivación que usa constantes de Henkin, cada aplicación de `forall_elim` en la derivación original debe seguir siendo válida después de reemplazar las constantes por variables frescas. Este lema asegura que la condición `IsFreeFor` (que es la guarda de `forall_elim`) se preserva a través de la transformación.

La condición `NoBoundVar w φ` es esencial: garantiza que la variable fresca w (que reemplaza a la constante) no es capturada por ningún cuantificador en φ. La condición $v \neq w$ evita que la variable de sustitución original coincida con la variable fresca, lo que podría causar una interferencia indeseada. Ambas condiciones se satisfacen automáticamente en la construcción de Henkin porque w se elige como una variable completamente nueva (mayor que todas las variables que aparecen en la derivación).

```
--
============================================================================
-- NO OCURRENCIA DE VARIABLES EN LA SUSTITUCIÓN DE HENKIN
--
============================================================================


lemma NotOccur_substHenkinTerm_var {s : Nat} (t : ArithTerm s) (c : HenkinConst)
(w : Nat) (n : Nat)
    (h_t : NotOccur n t) (h_nw : n ≠ w) :
    NotOccur n (substHenkinTerm c (ArithTerm.var w) t) := by
  induction t with
  | var _ => unfold substHenkinTerm; exact h_t
  | const _ => exact NotOccur.const _
  | henkin _ =>
    unfold substHenkinTerm
    split_ifs with _h_eq
    · exact NotOccur.var w h_nw
    · exact NotOccur.henkin _
  | zero => exact NotOccur.zero
  | succ _ ih =>
    cases h_t with | succ _ h_sub =>
    unfold substHenkinTerm; exact NotOccur.succ _ (ih h_sub)
  | add _ _ ih1 ih2 =>
    cases h_t with | add _ _ h1 h2 =>
    unfold substHenkinTerm; exact NotOccur.add _ _ (ih1 h1) (ih2 h2)
```

```
  | mul _ _ ih1 ih2 =>
    cases h_t with | mul _ _ h1 h2 =>
    unfold substHenkinTerm; exact NotOccur.mul _ _ (ih1 h1) (ih2 h2)
  | exp _ _ ih1 ih2 =>
    cases h_t with | exp _ _ h1 h2 =>
    unfold substHenkinTerm; exact NotOccur.exp _ _ (ih1 h1) (ih2 h2)
  | pred _ ih =>
    cases h_t with | pred _ h_sub =>
    unfold substHenkinTerm; exact NotOccur.pred _ (ih h_sub)

lemma IsFreeFor_substHenkin_var {s : Nat} (φ : ArithForm s) (v : Nat) (t : ArithTerm
s) (c : HenkinConst) (w : Nat)
    (_h_vw : v ≠ w) (h_bound : NoBoundVar w φ) (h : IsFreeFor v t φ) :
    IsFreeFor v (substHenkinTerm c (ArithTerm.var w) t) (substHenkinForm c
(ArithTerm.var w) φ) := by
  induction φ generalizing t with
  | bottom => trivial
  | eq _ _ => trivial
  | impl f1 f2 ih1 ih2 =>
    simp [substHenkinForm, IsFreeFor, ih1 t h_bound.1 h.1, ih2 t h_bound.2 h.2]
  | neg f ih =>
    simp [substHenkinForm, IsFreeFor, ih t h_bound h]
  | forall_form n f ih =>
    dsimp [IsFreeFor] at h
    unfold substHenkinForm
    dsimp [IsFreeFor]
    by_cases h_vn : v = n
    · subst h_vn; simp
    · simp [h_vn] at h
```

```
    rcases h with ⟨h_not, h_free⟩
    simp [h_vn]
    have h_nw : n ≠ w := Ne.symm h_bound.1
    exact ⟨NotOccur_substHenkinTerm_var t c w n h_not h_nw, ih t h_bound.2 h_free⟩
  | provable_op _ _ => trivial
  | true_op _ _ => trivial
  | list_nth_eq _ _ _ => trivial
```

Lemas de soporte para preservación de frescura y transformación de conjuntos

Se demuestran tres lemas que conectan las nociones de frescura en derivaciones con las propiedades sintácticas de no ocurrencia, y se establece una propiedad de distribución para la sustitución de Henkin sobre conjuntos.

Lema `NoBoundVar_of_NotOccurForm`: No ocurrencia implica no ligadura

Establece formalmente que si una variable no ocurre en absoluto en una fórmula, entonces no está ligada en ella:

$$ \forall w \in \mathbb{N},\ \forall \varphi \in \text{ArithForm } s,\ \text{NotOccurForm } w\ \varphi \implies \text{NoBoundVar } w\ \varphi $$

La demostración es por inducción sobre φ. El caso clave es `forall_form n f`: `NotOccurForm w` proporciona $w \neq n$ y `NotOccurForm w f`, y por hipótesis de inducción `NoBoundVar w f`. La conjunción de $w \neq n$ y `NoBoundVar w f` es exactamente `NoBoundVar w (forall_form n f)`.

Lema `FreshForDerivation_imp_NotOccurForm`: Las derivaciones frescas producen fórmulas sin la variable fresca

Establece la propiedad fundamental del predicado `FreshForDerivation`: si una derivación es fresca respecto a una variable w, entonces w no ocurre en la conclusión de la derivación:

$$ \forall d : \text{SPA_Deduction_Internal } s\ \Gamma\ \varphi,\ \text{FreshForDerivation } w\ d \implies \text{NotOccurForm } w\ \varphi $$

Estructura de la demostración

La prueba es por inducción sobre la derivación de `FreshForDerivation w h`, analizando cada regla de inferencia. Para cada constructor de `FreshForDerivation`, se extrae la información de frescura de las premisas y se construye `NotOccurForm w` para la conclusión:

- **`hyp`:** Directo de la hipótesis `h_w`.

- **`imp_intro`:** La conclusión es $\varphi \to \psi$. La frescura de φ y de ψ (por hipótesis de inducción) da `NotOccurForm w` para ambas, y por tanto para la implicación.

- **`forall_intro`:** La conclusión es $\forall v\ \varphi$. La hipótesis `hnw : v ≠ w` junto con `NotOccurForm w φ` (por hipótesis de inducción) dan `NotOccurForm w (forall_form v φ)`.

- **`forall_elim`:** La conclusión es $\varphi[v := t]$. La hipótesis de inducción da `NotOccurForm w φ`, y `h_wt` da `NotOccur w t`. Por `NotOccurForm_subst_other`, se obtiene `NotOccurForm w (substForm v t φ)`.

- **Reglas de igualdad:** Se construye `NotOccurForm w` para las conclusiones a partir de las hipótesis de frescura de los términos involucrados.

- **`ax_peano1`, `ax_peano2`, `induction`:** Se construyen directamente a partir de las condiciones de frescura de las premisas.

Lema `substHenkinSet_insert`: Distribución de la sustitución de Henkin sobre la inserción

Establece una propiedad de conmutación entre la sustitución de Henkin y la operación de inserción en conjuntos:

$$ \{\text{substHenkinForm } c\ u\ \psi \mid \psi \in \text{insert } \varphi\ \Gamma\} = \text{insert } (\text{substHenkinForm } c\ u\ \varphi)\ \{\text{substHenkinForm } c\ u\ \psi \mid \psi \in \Gamma\} $$

La demostración es por extensionalidad de conjuntos, analizando los casos de pertenencia a la inserción.

Significado

Estos lemas son piezas auxiliares para la gran demostración de eliminación de constantes de Henkin. `FreshForDerivation_imp_NotOccurForm` conecta el predicado inductivo sobre derivaciones con la propiedad sintáctica de no ocurrencia, permitiendo extraer información sintáctica de una derivación que se sabe fresca. `substHenkinSet_insert` es un lema de reescritura de conjuntos que facilita la manipulación de los contextos deductivos durante la transformación de derivaciones. `NoBoundVar_of_NotOccurForm` establece la jerarquía entre los predicados de frescura, confirmando que `NotOccurForm` es la noción más fuerte (implica tanto `NotFreeForm` como `NoBoundVar`).

```
--
============================================================

-- LEMAS ADICIONALES DE SOPORTE PARA PRESERVACIÓN DE FRESCURA

--
============================================================


lemma NoBoundVar_of_NotOccurForm {s : Nat} (w : Nat) (φ : ArithForm s) (h :
NotOccurForm w φ) :

   NoBoundVar w φ := by

 induction φ with

 | bottom => trivial
```

```
  | eq _ _ => trivial
  | impl f1 f2 ih1 ih2 =>
    exact ⟨ih1 h.1, ih2 h.2⟩
  | neg f ih =>
    exact ih h
  | forall_form n f ih =>
    exact ⟨h.1, ih h.2⟩
  | provable_op _ _ => trivial
  | true_op _ _ => trivial
  | list_nth_eq _ _ _ => trivial

lemma FreshForDerivation_imp_NotOccurForm {s : Nat} {Γ : Set (ArithForm s)} {φ :
ArithForm s}
    (w : Nat) (h : SPA_Deduction_Internal s Γ φ) (h_fresh : FreshForDerivation w h) :
    NotOccurForm w φ := by
  induction h_fresh with
  | hyp Γ' φ' h_in hw => exact hw
  | imp_intro Γ' φ' ψ' h' hwφ ih_h ih =>
    unfold NotOccurForm; exact ⟨hwφ, ih⟩
  | imp_elim Γ' φ' ψ' h1 h2 ih1_h ih2_h ih1 ih2 =>
    unfold NotOccurForm at ih1
    exact ih1.2
  | double_neg Γ' φ' h' ih_h ih =>
    unfold NotOccurForm at ih; exact ih
  | neg_intro Γ' φ' h' hwφ ih_h ih =>
    unfold NotOccurForm; exact hwφ
  | neg_elim Γ' φ' h1 h2 ih1_h ih2_h ih1 ih2 =>
    unfold NotOccurForm; trivial
```

```
| forall_intro Γ' v' φ' h' h_vΓ hnw ih_h ih =>
  unfold NotOccurForm; exact ⟨Ne.symm hnw, ih⟩
| forall_elim Γ' v' φ' t' h' h_free hnw h_wt ih_h ih =>
  have h_not : NotOccurForm w φ' := ih.2
  exact NotOccurForm_subst_other φ' v' w t' h_not h_wt
| eq_refl Γ' t' h_wt =>
  exact ⟨h_wt, h_wt⟩
| eq_symm Γ' t1 t2 h' ih_h ih =>
  exact ⟨ih.2, ih.1⟩
| eq_trans Γ' t1 t2 t3 h1 h2 ih1_h ih2_h ih1 ih2 =>
  exact ⟨ih1.1, ih2.2⟩
| eq_succ Γ' t1 t2 h' ih_h ih =>
  unfold NotOccurForm at ih ⊢
  exact ⟨NotOccur.succ t1 ih.1, NotOccur.succ t2 ih.2⟩
| eq_add Γ' t1 t2 u1 u2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  unfold NotOccurForm at ih1 ih2 ⊢
  exact ⟨NotOccur.add t1 u1 ih1.1 ih2.1, NotOccur.add t2 u2 ih1.2 ih2.2⟩
| eq_mul Γ' t1 t2 u1 u2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  unfold NotOccurForm at ih1 ih2 ⊢
  exact ⟨NotOccur.mul t1 u1 ih1.1 ih2.1, NotOccur.mul t2 u2 ih1.2 ih2.2⟩
| eq_exp Γ' t1 t2 u1 u2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  unfold NotOccurForm at ih1 ih2 ⊢
  exact ⟨NotOccur.exp t1 u1 ih1.1 ih2.1, NotOccur.exp t2 u2 ih1.2 ih2.2⟩
| eq_pred Γ' t1 t2 h' ih_h ih =>
  unfold NotOccurForm at ih ⊢
  exact ⟨NotOccur.pred t1 ih.1, NotOccur.pred t2 ih.2⟩
| eq_provable_op Γ' h_eq t1 t2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  unfold NotOccurForm at ih1 ih2 ⊢
```

```
  exact ih1.2
| eq_true_op Γ' h_eq t1 t2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  unfold NotOccurForm at ih1 ih2 ⊢
  exact ih1.2
| eq_list_nth_eq_1 Γ' t1 t2 t3 u1 h1 h2 ih1_h ih2_h ih1 ih2 =>
  unfold NotOccurForm at ih1 ih2 ⊢
  exact ⟨ih1.2, ih2.2.1, ih2.2.2⟩
| eq_list_nth_eq_2 Γ' t1 t2 t3 u2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  unfold NotOccurForm at ih1 ih2 ⊢
  exact ⟨ih2.1, ih1.2, ih2.2.2⟩
| eq_list_nth_eq_3 Γ' t1 t2 t3 u3 h1 h2 ih1_h ih2_h ih1 ih2 =>
  unfold NotOccurForm at ih1 ih2 ⊢
  exact ⟨ih2.1, ih2.2.1, ih1.2⟩
| eq_subst Γ' v' φ'' t1 t2 h1 h2 h_free1 h_free2 hnw h_wφ ih1_h ih2_h ih1 ih2 =>
  exact NotOccurForm_subst_other φ'' v' w t2 h_wφ ih1.2
| weaken Γ' φ' Γ'' h' h_sub ih_h ih => exact ih
| ax_peano1 Γ' x' h_wx =>
  unfold NotOccurForm; exact ⟨NotOccur.succ x' h_wx, NotOccur.zero⟩
| ax_peano2 Γ' x' y' h_wx h_wy =>
  unfold NotOccurForm
  have h1 : NotOccur w (ArithTerm.succ x') ∧ NotOccur w (ArithTerm.succ y') :=
    ⟨NotOccur.succ x' h_wx, NotOccur.succ y' h_wy⟩
  have h2 : NotOccur w x' ∧ NotOccur w y' := ⟨h_wx, h_wy⟩
  exact ⟨h1, h2⟩
| induction Γ' φ'' v' h1 h2 hnw h_wφ ih1_h ih2_h ih1 ih2 =>
  unfold NotOccurForm
  exact ⟨Ne.symm hnw, h_wφ⟩
```

```
lemma substHenkinSet_insert {s : Nat} (c : HenkinConst) (u : ArithTerm s) (φ : ArithForm s) (Γ : Set (ArithForm s)) :

  {x | ∃ ψ ∈ insert φ Γ, substHenkinForm c u ψ = x} = insert (substHenkinForm c u φ) {x | ∃ ψ ∈ Γ, substHenkinForm c u ψ = x} := by

 ext χ

 simp only [Set.mem_insert_iff]

 constructor

 · rintro ⟨ψ, h_or, rfl⟩

  cases h_or with

  | inl h1 => left; rw [h1]

  | inr h2 => right; exact ⟨ψ, h2, rfl⟩

 · rintro (rfl | ⟨ψ, h_ψ, rfl⟩)

  · exact ⟨φ, Or.inl rfl, rfl⟩

  · exact ⟨ψ, Or.inr h_ψ, rfl⟩
```

Teorema `substHenkin_preservation_of_fresh`: Preservación de la demostrabilidad bajo eliminación de constantes de Henkin

Se demuestra el teorema fundamental de eliminación de constantes de Henkin: si una fórmula φ es demostrable a partir de un conjunto de hipótesis Γ, y la derivación es fresca respecto a una variable w, entonces al reemplazar todas las

ocurrencias de una constante de Henkin c por $\text{var } w$, la fórmula transformada es demostrable a partir del conjunto de hipótesis transformado:

$$ \forall \Gamma \subseteq \text{ArithForm } s,\ \forall \varphi \in \text{ArithForm } s,\ \forall d : \Gamma \vdash \varphi, $$

$$ \forall w \in \mathbb{N},\ \text{FreshForDerivation } w\ d \implies \forall c \in \text{HenkinConst}, $$

$$ \{\text{substHenkinForm } c\ (\text{var } w)\ \psi \mid \psi \in \Gamma\} \vdash \text{substHenkinForm } c\ (\text{var } w)\ \varphi $$

Estructura de la demostración

La prueba procede por inducción sobre la derivación de `FreshForDerivation w h`, reconstruyendo una derivación en el sistema transformado para cada regla de inferencia. Es una demostración extensa que cubre más de treinta casos, pero cada uno sigue un patrón sistemático:

Reglas que no involucran variables ligadas ni sustituciones: La mayoría de las reglas (`hyp`, `imp_elim`, `double_neg`, `neg_elim`, reglas de igualdad, `weaken`, `ax_peano1`, `ax_peano2`) se transforman de manera directa: se aplica la misma regla a las derivaciones transformadas de las premisas. La hipótesis de inducción proporciona las derivaciones transformadas para las subderivaciones.

Reglas que modifican el contexto (`imp_intro`, `neg_intro`): Requieren manipular el conjunto de hipótesis transformado. El lema `substHenkinSet_insert` permite distribuir la transformación sobre la inserción, preservando la estructura del contexto.

`forall_intro`: Requiere verificar que la variable generalizada v' sigue sin ser libre en el contexto transformado. Esto se sigue del lema `NotFreeInSet_substHenkin`, utilizando que $v' \neq w$ (provisto por `FreshForDerivation`) y que `NotOccur v' (var w)` es cierto precisamente porque $v' \neq w$.

`forall_elim`: Es uno de los casos más delicados. Se debe verificar que:

1. La fórmula transformada es $\text{substHenkinForm } c\ (\text{var } w)\ \varphi'$.

2. El término transformado es $\text{substHenkinTerm } c\ (\text{var } w)\ t'$.

3. La condición `IsFreeFor` se preserva: `IsFreeFor_substHenkin_var` garantiza que el término transformado es libre para v' en la fórmula transformada.

4. La igualdad sintáctica: `substHenkinForm_substForm_var` permite conmutar la sustitución de Henkin con la sustitución de variable, obteniendo exactamente la conclusión requerida.

`eq_subst`: Similar a `forall_elim` en complejidad: requiere preservar las condiciones `IsFreeFor` para ambos términos t_1 y t_2, y utilizar la conmutación de sustituciones.

`induction`: Requiere transformar tanto el caso base ($\varphi[v := 0]$) como el paso inductivo ($\forall v\ (\varphi \to \varphi[v := S(v)]$). Para el caso base se utiliza la conmutación con `ArithTerm.zero`; para el paso inductivo se utiliza la conmutación con `ArithTerm.succ (ArithTerm.var v)`.

Significado

`substHenkin_preservation_of_fresh` es el teorema central de la Fase 8 y uno de los resultados más importantes de todo el desarrollo. Establece que la eliminación de constantes de Henkin es una transformación que preserva la demostrabilidad: cualquier demostración en la teoría expandida puede ser transformada en una demostración en la teoría original (módulo la sustitución de las constantes por variables frescas).

Este teorema es el análogo sintáctico del hecho semántico de que la extensión por constantes de Henkin es conservativa. La clave técnica es la noción de `FreshForDerivation`: la variable w utilizada para reemplazar la constante de Henkin debe ser completamente fresca respecto a toda la derivación, lo cual garantiza que no interfiere con las variables ligadas ni con las condiciones de libertad de las

reglas de inferencia. La construcción `exists_max_var_deriv` + `FreshForDerivation_of_gt_max_var` asegura que siempre es posible elegir una variable w suficientemente grande que satisfaga esta condición.

Este resultado completa la maquinaria necesaria para la prueba de completitud: dada una teoría consistente Γ, se construye su extensión de Henkin Γ_{Henkin} (añadiendo axiomas de testigo), se aplica Lindenbaum para obtener una extensión maximal consistente Δ, y finalmente se utiliza `substHenkin_preservation_of_fresh` para eliminar las constantes de Henkin y obtener un modelo a partir de la teoría original.

```
--
============================================================================

-- TEOREMA CRUCIAL: PRESERVACIÓN DE FRESCURA EN DERIVACIONES

--
============================================================================


theorem substHenkin_preservation_of_fresh {s : Nat} (Γ : Set (ArithForm s)) (φ :
ArithForm s)

  (h : SPA_Deduction_Internal s Γ φ) (w : Nat) (h_fresh : FreshForDerivation w h) (c :
HenkinConst) :

  SPA_Deduction_Internal s {x | ∃ ψ ∈ Γ, substHenkinForm c (ArithTerm.var w) ψ = x}
(substHenkinForm c (ArithTerm.var w) φ) := by

 induction h_fresh generalizing c with

 | hyp Γ' φ' h_in h_w =>

  apply SPA_Deduction_Internal.hyp
```

```
    exact ⟨φ', h_in, rfl⟩
  | imp_intro Γ' φ' ψ' h' h_wφ ih_h ih =>
    have ih_c := ih c
    rw [substHenkinSet_insert] at ih_c
    exact SPA_Deduction_Internal.imp_intro _ _ _ ih_c
  | imp_elim Γ' φ' ψ' h1 h2 ih1_h ih2_h ih1 ih2 =>
    exact SPA_Deduction_Internal.imp_elim _ (substHenkinForm c (ArithTerm.var w) φ') (substHenkinForm c (ArithTerm.var w) ψ') (ih1 c) (ih2 c)
  | double_neg Γ' φ' h' ih_h ih =>
    exact SPA_Deduction_Internal.double_neg _ _ (ih c)
  | neg_intro Γ' φ' h' h_wφ ih_h ih =>
    have ih_c := ih c
    rw [substHenkinSet_insert] at ih_c
    exact SPA_Deduction_Internal.neg_intro _ _ ih_c
  | neg_elim Γ' φ' h1 h2 ih1_h ih2_h ih1 ih2 =>
    exact SPA_Deduction_Internal.neg_elim _ (substHenkinForm c (ArithTerm.var w) φ') (ih1 c) (ih2 c)
  | forall_intro Γ' v' φ' h' h_vΓ hnw ih_h ih =>
    have h_vΓ' : NotFreeInSet v' {x | ∃ ψ ∈ Γ', substHenkinForm c (ArithTerm.var w) ψ = x} := by
      apply NotFreeInSet_substHenkin
      · exact h_vΓ
      · exact NotOccur.var w hnw
    exact SPA_Deduction_Internal.forall_intro _ v' _ (ih c) h_vΓ'
  | forall_elim Γ' v' φ' t' h' h_free hnw h_wt ih_h ih =>
    have h_deriv := ih c
    have h_not : NotOccurForm w (ArithForm.forall_form v' φ') := FreshForDerivation_imp_NotOccurForm w h' ih_h
    have hw_phi : NotOccurForm w φ' := h_not.2
    have h_bound : NoBoundVar w φ' := NoBoundVar_of_NotOccurForm w φ' hw_phi
```

```
  have h_free' : IsFreeFor v' (substHenkinTerm c (ArithTerm.var w) t') (substHenkinForm c (ArithTerm.var w) φ') :=
    IsFreeFor_substHenkin_var φ' v' t' c w hnw h_bound h_free
  have h_elim := SPA_Deduction_Internal.forall_elim _ v' (substHenkinForm c (ArithTerm.var w) φ') (substHenkinTerm c (ArithTerm.var w) t') h_deriv h_free'
  rw [substHenkinForm_substForm_var φ' c w v' t' hnw]
  exact h_elim
| eq_refl Γ' t' h_wt =>
  exact SPA_Deduction_Internal.eq_refl _ _
| eq_symm Γ' t1 t2 h' ih_h ih =>
  exact SPA_Deduction_Internal.eq_symm _ _ _ (ih c)
| eq_trans Γ' t1 t2 t3 h1 h2 ih1_h ih2_h ih1 ih2 =>
  exact SPA_Deduction_Internal.eq_trans _ _ _ _ (ih1 c) (ih2 c)
| eq_succ Γ' t1 t2 h' ih_h ih =>
  exact SPA_Deduction_Internal.eq_succ _ _ _ (ih c)
| eq_add Γ' t1 t2 u1 u2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  exact SPA_Deduction_Internal.eq_add _ _ _ _ _ (ih1 c) (ih2 c)
| eq_mul Γ' t1 t2 u1 u2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  exact SPA_Deduction_Internal.eq_mul _ _ _ _ _ (ih1 c) (ih2 c)
| eq_exp Γ' t1 t2 u1 u2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  exact SPA_Deduction_Internal.eq_exp _ _ _ _ _ (ih1 c) (ih2 c)
| eq_pred Γ' t1 t2 h' ih_h ih =>
  exact SPA_Deduction_Internal.eq_pred _ _ _ (ih c)
| eq_provable_op Γ' h_eq t1 t2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  exact SPA_Deduction_Internal.eq_provable_op _ h_eq _ _ (ih1 c) (ih2 c)
| eq_true_op Γ' h_eq t1 t2 h1 h2 ih1_h ih2_h ih1 ih2 =>
  exact SPA_Deduction_Internal.eq_true_op _ h_eq _ _ (ih1 c) (ih2 c)
| eq_list_nth_eq_1 Γ' t1 t2 t3 u1 h1 h2 ih1_h ih2_h ih1 ih2 =>
  exact SPA_Deduction_Internal.eq_list_nth_eq_1 _ _ _ _ _ (ih1 c) (ih2 c)
```

```
  | eq_list_nth_eq_2 Γ' t1 t2 t3 u2 h1 h2 ih1_h ih2_h ih1 ih2 =>
    exact SPA_Deduction_Internal.eq_list_nth_eq_2 _ _ _ _ _ (ih1 c) (ih2 c)
  | eq_list_nth_eq_3 Γ' t1 t2 t3 u3 h1 h2 ih1_h ih2_h ih1 ih2 =>
    exact SPA_Deduction_Internal.eq_list_nth_eq_3 _ _ _ _ _ (ih1 c) (ih2 c)
  | eq_subst Γ' v' φ'' t1 t2 h1 h2 h_free1 h_free2 hnw h_wφ ih1_h ih2_h ih1 ih2 =>
    have h_eq' := ih1 c
    have h_sub' := ih2 c
    rw [substHenkinForm_substForm_var φ'' c w v' t1 hnw] at h_sub'
    have h_bound : NoBoundVar w φ'' := NoBoundVar_of_NotOccurForm w φ'' h_wφ
    have h_free1' : IsFreeFor v' (substHenkinTerm c (ArithTerm.var w) t1)
(substHenkinForm c (ArithTerm.var w) φ'') :=
      IsFreeFor_substHenkin_var φ'' v' t1 c w hnw h_bound h_free1
    have h_free2' : IsFreeFor v' (substHenkinTerm c (ArithTerm.var w) t2)
(substHenkinForm c (ArithTerm.var w) φ'') :=
      IsFreeFor_substHenkin_var φ'' v' t2 c w hnw h_bound h_free2
    have h_deriv := SPA_Deduction_Internal.eq_subst _ v' (substHenkinForm c
(ArithTerm.var w) φ'') (substHenkinTerm c (ArithTerm.var w) t1) (substHenkinTerm
c (ArithTerm.var w) t2) h_eq' h_sub' h_free1' h_free2'
    rw [substHenkinForm_substForm_var φ'' c w v' t2 hnw]
    exact h_deriv
  | weaken Γ' φ' Γ'' h' h_sub ih_h ih =>
    have h_sub_image : {x | ∃ ψ ∈ Γ', substHenkinForm c (ArithTerm.var w) ψ = x} ⊆ {x |
∃ ψ ∈ Γ'', substHenkinForm c (ArithTerm.var w) ψ = x} := by
      intro ψ' h_mem
      rcases h_mem with ⟨ψ, h_ψ, rfl⟩
      exact ⟨ψ, h_sub h_ψ, rfl⟩
    exact SPA_Deduction_Internal.weaken _ _ _ (ih c) h_sub_image
  | ax_peano1 Γ' x' h_wx =>
    exact SPA_Deduction_Internal.ax_peano1 _ _
  | ax_peano2 Γ' x' y' h_wx h_wy =>
```

```
  exact SPA_Deduction_Internal.ax_peano2 _ _ _

| induction Γ'' φ'' v' h1 h2 hnw h_wφ ih1_h ih2_h ih1 ih2 =>

  have h_ind1 := ih1 c

  rw [substHenkinForm_substForm_var φ'' c w v' ArithTerm.zero hnw] at h_ind1

  have h_ind2 := ih2 c

  have h_comm : substHenkinForm c (ArithTerm.var w) (substForm v'
(ArithTerm.succ (ArithTerm.var v')) φ'') =

        substForm v' (ArithTerm.succ (ArithTerm.var v')) (substHenkinForm c
(ArithTerm.var w) φ'') := by

   rw [substHenkinForm_substForm_var φ'' c w v' (ArithTerm.succ (ArithTerm.var
v')) hnw]

   rfl

  change SPA_Deduction_Internal s {x | ∃ ψ ∈ Γ'', substHenkinForm c (ArithTerm.var
w) ψ = x}

      (ArithForm.forall_form v' (ArithForm.impl (substHenkinForm c (ArithTerm.var
w) φ'') (substHenkinForm c (ArithTerm.var w) (substForm v' (ArithTerm.succ
(ArithTerm.var v')) φ'')))) at h_ind2

  rw [h_comm] at h_ind2

  exact SPA_Deduction_Internal.induction _ (substHenkinForm c (ArithTerm.var w)
φ'') v' h_ind1 h_ind2
```

Lemas de conmutación y eliminación de constantes de Henkin

Se demuestran cuatro lemas que establecen propiedades de conmutación entre la sustitución de constantes de Henkin y la sustitución de variables cuando el término sustituido es precisamente la constante que se elimina. Estos resultados son el núcleo técnico de la eliminación de constantes de Henkin en el contexto de los axiomas de testigo.

Lema `substHenkinTerm_substTerm_henkin_comm`: Conmutación general para términos

Establece que sustituir primero una variable v por la constante de Henkin c y luego eliminar c reemplazándola por $\text{var } w$ es equivalente a eliminar primero c y luego sustituir v por $\text{var } w$:

$$ \forall t \in \text{ArithTerm } s,\ \forall c \in \text{HenkinConst},\ \forall w, v \in \mathbb{N}, $$

$$ \text{substHenkinTerm } c\ (\text{var } w)\ (\text{substTerm } v\ (\text{henkin } c)\ t) = \text{substTerm } v\ (\text{var } w)\ (\text{substHenkinTerm } c\ (\text{var } w)\ t) $$

Estructura de la demostración

Por inducción sobre t:

- **`var n`:** Si $n = v$, el lado izquierdo reemplaza $\text{var } v$ por $\text{henkin } c$ y luego c por $\text{var } w$, resultando en $\text{var } w$. El lado derecho

elimina c primero (no afecta a $\text{var } v$) y luego reemplaza v por $\text{var } w$, también resultando en $\text{var } w$.

- **`henkin c'`:** Si $c' = c$, ambos lados reemplazan la constante por $\text{var } w$ (en el lado izquierdo, la primera sustitución no afecta porque es sobre v, no sobre constantes; la segunda elimina c).

- **Casos recursivos:** Se aplican las hipótesis de inducción.

Lema `substHenkinForm_substForm_henkin_comm`: Conmutación general para fórmulas

Extiende el resultado anterior al nivel de fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall c \in \text{HenkinConst},\ \forall w, v \in \mathbb{N}, $$

$$ \text{substHenkinForm } c\ (\text{var } w)\ (\text{substForm } v\ (\text{henkin } c)\ \varphi) = \text{substForm } v\ (\text{var } w)\ (\text{substHenkinForm } c\ (\text{var } w)\ \varphi) $$

La demostración es por inducción sobre φ, utilizando `substHenkinTerm_substTerm_henkin_comm` para los casos con términos. Para `forall_form n f`, se distingue si $v = n$ (la sustitución no penetra) o no.

Lema `substHenkinTerm_substTerm_henkin_self`: Simplificación cuando la constante no ocurre

Si la constante c no ocurre en t, entonces el resultado de la doble operación (sustituir v por c y luego eliminar c) es simplemente sustituir v por $\text{var } w$ en t:

$$ \text{ConstantNotOccur } c\ t \implies \text{substHenkinTerm } c\ (\text{var } w)\ (\text{substTerm } v\ (\text{henkin } c)\ t) = \text{substTerm } v\ (\text{var } w)\ t $$

La demostración utiliza el lema de conmutación general y luego `substHenkinTerm_fresh` para eliminar la sustitución de Henkin (ya que c no ocurre en t).

Lema `substHenkinForm_substForm_henkin_self`: Simplificación en fórmulas

Versión análoga para fórmulas:

$$ \text{ConstantNotOccurForm } c\ \varphi \implies \text{substHenkinForm } c\ (\text{var } w)\ (\text{substForm } v\ (\text{henkin } c)\ \varphi) = \text{substForm } v\ (\text{var } w)\ \varphi $$

Significado

Estos lemas de conmutación capturan la esencia de la eliminación de constantes de Henkin en el contexto de los axiomas de testigo. Recordemos que un axioma de testigo tiene la forma:

$$ \text{freshEx} \to \psi[v := \text{henkin } c] $$

donde `freshEx` no contiene a c (por construcción). Al eliminar la constante c (reemplazándola por una variable fresca w), el consecuente $\psi[v := \text{henkin } c]$ se transforma, por los lemas de conmutación, en $(\text{substHenkinForm } c\ (\text{var } w)\ \psi)[v := \text{var } w]$. Si además ψ no contiene a c (lo cual

es cierto para la fórmula original antes de añadir el testigo), esto se simplifica a $\psi[v := \text{var } w]$, que es exactamente `freshEx` (por definición de `freshEx`).

Esta cadena de transformaciones muestra que el axioma de testigo, tras eliminar las constantes de Henkin, se convierte en `freshEx → freshEx`, una tautología. Esto explica por qué la extensión de Henkin es conservativa: los nuevos axiomas añadidos se vuelven triviales una vez que las constantes de Henkin son eliminadas. Los lemas de conmutación proporcionan la justificación sintáctica formal de esta intuición.

```
--
================================================================

-- CONMUTACIÓN GENERAL Y ELIMINACIÓN DE CONSTANTES DE HENKIN

--
================================================================


lemma substHenkinTerm_substTerm_henkin_comm {s : Nat} (t : ArithTerm s) (c :
HenkinConst) (w : Nat) (v : Nat) :

  substHenkinTerm c (ArithTerm.var w) (substTerm v (ArithTerm.henkin c) t) =
substTerm v (ArithTerm.var w) (substHenkinTerm c (ArithTerm.var w) t) := by

 induction t with

 | var n =>

  dsimp [substTerm, substHenkinTerm]
```

```
  split_ifs with h_vn
  · dsimp [substHenkinTerm]
    rw [if_pos rfl]
  · rfl
| const c' => rfl
| henkin c' =>
  dsimp [substTerm, substHenkinTerm]
  split_ifs with h_eq
  · subst h_eq
    dsimp [substTerm]
    split_ifs <;> rfl
  · rfl
| zero => rfl
| succ t' ih =>
  dsimp [substTerm, substHenkinTerm]
  rw [ih]
| add t1 t2 ih1 ih2 =>
  dsimp [substTerm, substHenkinTerm]
  rw [ih1, ih2]
| mul t1 t2 ih1 ih2 =>
  dsimp [substTerm, substHenkinTerm]
  rw [ih1, ih2]
| exp t1 t2 ih1 ih2 =>
  dsimp [substTerm, substHenkinTerm]
  rw [ih1, ih2]
| pred t' ih =>
  dsimp [substTerm, substHenkinTerm]
  rw [ih]
```

```
lemma substHenkinForm_substForm_henkin_comm {s : Nat} (φ : ArithForm s) (c : HenkinConst) (w : Nat) (v : Nat) :
  substHenkinForm c (ArithTerm.var w) (substForm v (ArithTerm.henkin c) φ) = substForm v (ArithTerm.var w) (substHenkinForm c (ArithTerm.var w) φ) := by
 induction φ with
 | bottom => rfl
 | eq t1 t2 =>
  dsimp [substForm, substHenkinForm]
  rw [substHenkinTerm_substTerm_henkin_comm t1 c w v, substHenkinTerm_substTerm_henkin_comm t2 c w v]
 | impl f1 f2 ih1 ih2 =>
  dsimp [substForm, substHenkinForm]
  rw [ih1, ih2]
 | neg f ih =>
  dsimp [substForm, substHenkinForm]
  rw [ih]
 | forall_form n f ih =>
  dsimp [substHenkinForm, substForm]
  split_ifs with h_vn
  · rfl
  · dsimp [substHenkinForm, substForm]
    rw [ih]
 | provable_op heq t =>
  dsimp [substForm, substHenkinForm]
  rw [substHenkinTerm_substTerm_henkin_comm t c w v]
 | true_op heq t =>
  dsimp [substForm, substHenkinForm]
  rw [substHenkinTerm_substTerm_henkin_comm t c w v]
```

```
  | list_nth_eq t1 t2 t3 =>

    dsimp [substForm, substHenkinForm]

    rw [substHenkinTerm_substTerm_henkin_comm t1 c w v,
substHenkinTerm_substTerm_henkin_comm t2 c w v,
substHenkinTerm_substTerm_henkin_comm t3 c w v]


lemma substHenkinTerm_substTerm_henkin_self {s : Nat} (t : ArithTerm s) (c :
HenkinConst) (w : Nat) (v : Nat) (h : ConstantNotOccur c t) :

    substHenkinTerm c (ArithTerm.var w) (substTerm v (ArithTerm.henkin c) t) =
substTerm v (ArithTerm.var w) t := by

  rw [substHenkinTerm_substTerm_henkin_comm, substHenkinTerm_fresh t c
(ArithTerm.var w) h]


lemma substHenkinForm_substForm_henkin_self {s : Nat} (φ : ArithForm s) (c :
HenkinConst) (w : Nat) (v : Nat) (h : ConstantNotOccurForm c φ) :

    substHenkinForm c (ArithTerm.var w) (substForm v (ArithTerm.henkin c) φ) =
substForm v (ArithTerm.var w) φ := by

  rw [substHenkinForm_substForm_henkin_comm, substHenkinForm_fresh φ c
(ArithTerm.var w) h]
```

Lema `substHenkinSet_insert_henkin`: Transformación del conjunto al eliminar una constante de Henkin

Se demuestra un lema crucial sobre la transformación de conjuntos de fórmulas bajo la eliminación de constantes de Henkin, específicamente para el caso en que el conjunto contiene un axioma de testigo.

Enunciado

Sea Γ un conjunto de fórmulas que no contienen una constante de Henkin c (es decir, `ConstantNotOccurSet c Γ`). Al aplicar la transformación de eliminación de c (reemplazándola por $\text{var } w$) al conjunto que resulta de insertar el axioma de testigo:

$$ \text{freshEx} \to \psi'[v := \text{henkin } c] $$

en Γ, se obtiene exactamente el conjunto Γ original (inalterado) más el axioma transformado:

$$ \text{freshEx}' \to (\text{substHenkinForm } c\ (\text{var } w)\ \psi')[v := \text{var } w] $$

donde $\text{freshEx}' = \neg \forall v\ \neg (\text{substHenkinForm } c\ (\text{var } w)\ \psi')$.

Estructura de la demostración

1. Distribución de la transformación sobre la inserción: Se aplica `substHenkinSet_insert` para separar el axioma de testigo del resto del conjunto Γ.

2. Simplificación de Γ: Dado que ninguna fórmula de Γ contiene la constante c, la transformación `substHenkinForm c (var w)` es la identidad sobre Γ. Esto se demuestra por extensionalidad: para cada χ, si proviene de alguna $\psi \in \Gamma$, entonces `substHenkinForm_fresh` garantiza que $\text{substHenkinForm } c\ (\text{var } w)\ \psi = \psi$, por lo que

$\chi = \psi \in \Gamma$. Recíprocamente, cada $\chi \in \Gamma$ es imagen de sí misma. Así, $\{\text{substHenkinForm } c\ (\text{var } w)\ \psi \mid \psi \in \Gamma\} = \Gamma$.

3. Transformación del axioma: Se aplica la definición de `substHenkinForm` al axioma. La parte `freshEx` se transforma reemplazando ψ' por $\text{substHenkinForm } c\ (\text{var } w)\ \psi'$. La parte del consecuente $\psi'[v := \text{henkin } c]$ se transforma usando el lema de conmutación `substHenkinForm_substForm_henkin_comm`, que da:

$$ \text{substHenkinForm } c\ (\text{var } w)\ (\text{substForm } v\ (\text{henkin } c)\ \psi') = \text{substForm } v\ (\text{var } w)\ (\text{substHenkinForm } c\ (\text{var } w)\ \psi') $$

Esto es exactamente la forma requerida para el consecuente transformado.

Significado

Este lema muestra de manera precisa qué ocurre con el conjunto de hipótesis al aplicar la eliminación de constantes de Henkin a una teoría que ha sido extendida con un axioma de testigo. El resultado es notable: el conjunto transformado consiste en el conjunto original Γ (sin modificar, porque no contenía la constante eliminada) más una nueva fórmula que es esencialmente la misma que el axioma original pero con la constante de Henkin reemplazada por la variable fresca w.

Este comportamiento es exactamente lo que se necesita para la demostración de completitud. En el proceso de eliminación de constantes de Henkin aplicado a una derivación en la teoría expandida, cada aparición del axioma de testigo en el conjunto de hipótesis se transforma en una fórmula que ya no contiene constantes de Henkin. Además, el consecuente transformado $\psi'[v := \text{var } w]$ es precisamente la negación de `freshEx'`, lo que hace que el axioma transformado sea equivalente a `freshEx' → freshEx'`, una tautología. Esto implica que los axiomas de testigo, tras la eliminación de constantes, se vuelven demostrables sin necesidad de ser incluidos como hipótesis, lo que permite "descargarlos" y obtener una derivación en la teoría original.

El lema también depende crucialmente de la hipótesis `ConstantNotOccurSet c Γ`, que garantiza que la constante eliminada no aparecía en la teoría original. Esta hipótesis se satisface en la construcción de Henkin porque las constantes se introducen progresivamente y siempre son frescas respecto a las fórmulas previas.

```
--
============================================================

-- RELACIONES DE CONJUNTOS BAJO SUSTITUCIÓN DE HENKIN

--
============================================================


lemma substHenkinSet_insert_henkin {s : Nat} (c : HenkinConst) (w : Nat) (v : Nat) (ψ'
: ArithForm s) (Γ : Set (ArithForm s)) (h_fresh : ConstantNotOccurSet c Γ) :

  {x | ∃ ψ ∈ insert (ArithForm.impl (ArithForm.neg (ArithForm.forall_form v
(ArithForm.neg ψ'))) (substForm v (ArithTerm.henkin c) ψ')) Γ, substHenkinForm c
(ArithTerm.var w) ψ = x} =

  insert (ArithForm.impl (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg
(substHenkinForm c (ArithTerm.var w) ψ')))) (substForm v (ArithTerm.var w)
(substHenkinForm c (ArithTerm.var w) ψ'))) Γ := by

 rw [substHenkinSet_insert]

 have h_Γ_eq : {x | ∃ ψ ∈ Γ, substHenkinForm c (ArithTerm.var w) ψ = x} = Γ := by

  ext χ

  constructor

  · rintro ⟨ψ, h_ψ, rfl⟩

   have h_co := h_fresh ψ h_ψ

   rw [substHenkinForm_fresh ψ c (ArithTerm.var w) h_co]
```

```
    exact h_ψ
  · intro h_χ
    use χ
    have h_co := h_fresh χ h_χ
    rw [substHenkinForm_fresh χ c (ArithTerm.var w) h_co]
    exact ⟨h_χ, rfl⟩
rw [h_Γ_eq]
simp only [substHenkinForm]
rw [substHenkinForm_substForm_henkin_comm]
```

Lemas lógicos adicionales y preservación de consistencia local

Se presentan tres resultados deductivos auxiliares y el teorema fundamental que garantiza que la extensión de Henkin preserva la consistencia localmente (al añadir un solo axioma de testigo).

Lema `ex_falso`: Principio de explosión

Establece que a partir de una contradicción se puede deducir cualquier fórmula:

$$ \forall \Gamma \subseteq \text{ArithForm } s,\ \forall \varphi \in \text{ArithForm } s,\ \Gamma \vdash \bot \implies \Gamma \vdash \varphi $$

La demostración utiliza `weaken` para llevar $\bot$ al contexto extendido con $\neg\varphi$, luego `neg_intro` para obtener $\neg\neg\varphi$, y finalmente `double_neg` para obtener φ.

Lema `imp_neg_elim_left`: Extracción del antecedente de una implicación negada

Establece que de $\neg(A \to B)$ se puede deducir A:

$$ \Gamma \vdash \neg(A \to B) \implies \Gamma \vdash A $$

La demostración procede por reducción al absurdo: se asume $\neg A$, y en el contexto extendido con A y $\neg A$ se deriva $\bot$, de donde por explosión se obtiene B. Por `imp_intro` se obtiene $A \to B$, que junto con $\neg(A \to B)$ produce una contradicción, forzando $\neg\neg A$ y por tanto A.

Lema `imp_neg_elim_right`: Extracción de la negación del consecuente

Establece que de $\neg(A \to B)$ se puede deducir $\neg B$:

$$ \Gamma \vdash \neg(A \to B) \implies \Gamma \vdash \neg B $$

La demostración es más directa: en el contexto extendido con B, a partir de A (obtenido por `imp_neg_elim_left`) y B se obtiene $A \to B$, que contradice $\neg(A \to B)$, forzando $\neg B$.

Teorema `consistency_preservation_local`: Preservación local de la consistencia

Establece el resultado clave de la construcción de Henkin: si Γ es un conjunto consistente que no contiene la constante de Henkin c, y ψ es una fórmula que no contiene c, entonces añadir el axioma de testigo para $\exists v\ \psi$ (usando la constante fresca c) preserva la consistencia:

$$ \text{Consistent } \Gamma \implies \text{Consistent}(\Gamma \cup \{\text{freshEx} \to \psi[v := \text{henkin } c]\}) $$

bajo las hipótesis adicionales de que cualquier derivación de $\bot$ en la teoría expandida es fresca respecto a v, y que v no es libre en Γ.

Estructura de la demostración

Se asume que la teoría expandida es inconsistente, es decir, existe una derivación h_{bot} de $\bot$ a partir de $\Gamma \cup \{\text{axioma}\}$. Se construye una derivación de $\bot$ en la teoría original Γ, contradiciendo su consistencia:

1. Aplicación de la eliminación de constantes: Por `substHenkin_preservation_of_fresh`, la derivación h_{bot} se transforma en una derivación en el conjunto transformado, reemplazando c por $\text{var } v$ (nótese que aquí se usa la misma variable v del cuantificador existencial, lo cual es posible por la hipótesis de frescura).

2. Simplificación del conjunto transformado: Por `substHenkinSet_insert_henkin`, el conjunto transformado es $\Gamma \cup \{\text{freshEx}' \to \psi[v := \text{var } v]\}$, donde $\text{freshEx}' = \neg \forall v\ \neg \psi$ (ya que ψ no contiene c, por lo que `substHenkinForm` es la identidad sobre ψ).

3. Simplificación del consecuente: $\psi[v := \text{var } v] = \psi$ por `substForm_self_var`. Por tanto, el axioma transformado es $\neg \forall v\ \neg \psi \to \psi$.

4. Análisis deductivo: De la derivación transformada (en $\Gamma \cup \{\neg \forall v\ \neg \psi \to \psi\} \vdash \bot$) se aplica `neg_intro` para obtener $\Gamma \vdash \neg(\neg \forall v\ \neg \psi \to \psi)$.

5. Descomposición de la negación: Por `imp_neg_elim_left`, $\Gamma \vdash \neg \forall v\ \neg \psi$ (el antecedente). Por `imp_neg_elim_right`, $\Gamma \vdash \neg \psi$ (la negación del consecuente).

6. Generalización y contradicción: Dado que v no es libre en Γ (`h_fresh_w_Gamma`), se aplica `forall_intro` a $\Gamma \vdash \neg \psi$ para obtener $\Gamma \vdash \forall v\ \neg \psi$. Pero también se tiene $\Gamma \vdash \neg \forall v\ \neg \psi$, lo que por `neg_elim` produce $\Gamma \vdash \bot$, contradiciendo `Consistent Γ`.

Significado

`consistency_preservation_local` es el resultado que justifica el paso inductivo en la construcción de Henkin. Muestra que añadir un axioma de testigo (con una constante de Henkin fresca) a una teoría consistente no introduce inconsistencias. La demostración utiliza de manera esencial toda la maquinaria de eliminación de constantes de Henkin desarrollada en las fases anteriores: `substHenkin_preservation_of_fresh` para transformar la derivación, `substHenkinSet_insert_henkin` para simplificar el contexto, y los lemas de conmutación y frescura para garantizar que las transformaciones son correctas.

La hipótesis `h_fresh_w_deriv` (que toda derivación de $\bot$ en la teoría expandida es fresca respecto a v) es técnica pero se satisface automáticamente cuando v se elige como una variable suficientemente grande (mayor que todas las variables que aparecen en la derivación), lo cual es siempre posible por `exists_max_var_deriv` y `FreshForDerivation_of_gt_max_var`. Esto cierra el círculo del argumento de Henkin: en cada paso, se puede elegir una variable y una constante de Henkin suficientemente grandes para garantizar tanto la frescura como la preservación de la consistencia.

```
--
================================================================

-- LEMAS DE IDENTIDAD PARA LA SUSTITUCIÓN DE VARIABLES

--
================================================================
```

```
--
================================================================

-- FASE 9: LEMAS LÓGICOS ADICIONALES Y PRESERVACIÓN DE CONSISTENCIA LOCAL

--
================================================================
```

```
lemma ex_falso {s : Nat} (Γ : Set (ArithForm s)) (φ : ArithForm s) (h : SPA_Deduction_Internal s Γ ArithForm.bottom) :

  SPA_Deduction_Internal s Γ φ := by

 have h1 : SPA_Deduction_Internal s (insert (ArithForm.neg φ) Γ) ArithForm.bottom :=

  SPA_Deduction_Internal.weaken Γ ArithForm.bottom (insert (ArithForm.neg φ) Γ) h (Set.subset_insert _ _)
```

```
  have h2 : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.neg φ)) :=
    SPA_Deduction_Internal.neg_intro Γ (ArithForm.neg φ) h1
  exact SPA_Deduction_Internal.double_neg Γ φ h2


lemma imp_neg_elim_left {s : Nat} (Γ : Set (ArithForm s)) (A B : ArithForm s)
    (h : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.impl A B))) :
    SPA_Deduction_Internal s Γ A := by
  have h_not_A : SPA_Deduction_Internal s (insert (ArithForm.neg A) Γ)
(ArithForm.neg A) :=
    SPA_Deduction_Internal.hyp _ _ (Set.mem_insert _ _)
  have h_A : SPA_Deduction_Internal s (insert A (insert (ArithForm.neg A) Γ)) A :=
    SPA_Deduction_Internal.hyp _ _ (Set.mem_insert _ _)
  have h_not_A_weak : SPA_Deduction_Internal s (insert A (insert (ArithForm.neg A)
Γ)) (ArithForm.neg A) :=
    SPA_Deduction_Internal.weaken (insert (ArithForm.neg A) Γ) (ArithForm.neg A) _
h_not_A (Set.subset_insert _ _)
  have h_bot_weak : SPA_Deduction_Internal s (insert A (insert (ArithForm.neg A) Γ))
ArithForm.bottom :=
    SPA_Deduction_Internal.neg_elim _ A h_A h_not_A_weak
  have h_B : SPA_Deduction_Internal s (insert A (insert (ArithForm.neg A) Γ)) B :=
    ex_falso _ B h_bot_weak
  have h_imp : SPA_Deduction_Internal s (insert (ArithForm.neg A) Γ) (ArithForm.impl
A B) :=
    SPA_Deduction_Internal.imp_intro _ A B h_B
  have h_neg_imp : SPA_Deduction_Internal s (insert (ArithForm.neg A) Γ)
(ArithForm.neg (ArithForm.impl A B)) :=
    SPA_Deduction_Internal.weaken Γ _ _ h (Set.subset_insert _ _)
  have h_bot : SPA_Deduction_Internal s (insert (ArithForm.neg A) Γ)
ArithForm.bottom :=
    SPA_Deduction_Internal.neg_elim _ _ h_imp h_neg_imp
  have h_nnA : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.neg A)) :=
```

```
  SPA_Deduction_Internal.neg_intro Γ (ArithForm.neg A) h_bot
 exact SPA_Deduction_Internal.double_neg Γ A h_nnA

lemma imp_neg_elim_right {s : Nat} (Γ : Set (ArithForm s)) (A B : ArithForm s)
  (h : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.impl A B))) :
  SPA_Deduction_Internal s Γ (ArithForm.neg B) := by
 have h_B : SPA_Deduction_Internal s (insert A (insert B Γ)) B :=
  SPA_Deduction_Internal.hyp _ B (Set.mem_insert_of_mem A (Set.mem_insert B Γ))
 have h_imp : SPA_Deduction_Internal s (insert B Γ) (ArithForm.impl A B) :=
  SPA_Deduction_Internal.imp_intro _ A B h_B
 have h_neg_imp : SPA_Deduction_Internal s (insert B Γ) (ArithForm.neg
(ArithForm.impl A B)) :=
  SPA_Deduction_Internal.weaken Γ _ _ h (Set.subset_insert B Γ)
 have h_bot : SPA_Deduction_Internal s (insert B Γ) ArithForm.bottom :=
  SPA_Deduction_Internal.neg_elim _ _ h_imp h_neg_imp
 exact SPA_Deduction_Internal.neg_intro Γ B h_bot

theorem consistency_preservation_local {s : Nat} (Γ : Set (ArithForm s)) (v : Nat) (ψ :
ArithForm s) (c : HenkinConst)
  (h_fresh_w_deriv : ∀ (h_bot : SPA_Deduction_Internal s (insert (ArithForm.impl
(ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) (substForm v
(ArithTerm.henkin c) ψ)) Γ) ArithForm.bottom), FreshForDerivation v h_bot)
  (h_fresh_w_Gamma : NotFreeInSet v Γ)
  (h_fresh_c_Gamma : ConstantNotOccurSet c Γ)
  (h_fresh_c_psi : ConstantNotOccurForm c ψ)
  (h_con : Consistent Γ) :
  Consistent (insert (ArithForm.impl (ArithForm.neg (ArithForm.forall_form v
(ArithForm.neg ψ))) (substForm v (ArithTerm.henkin c) ψ)) Γ) := by
 intro h_bot
 have h_fresh := h_fresh_w_deriv h_bot
```

have h_subst := substHenkin_preservation_of_fresh _ _ h_bot v h_fresh c

have h_set := substHenkinSet_insert_henkin c v v ψ Γ h_fresh_c_Gamma

have h_psi_eq : substHenkinForm c (ArithTerm.var v) ψ = ψ := substHenkinForm_fresh ψ c (ArithTerm.var v) h_fresh_c_psi

rw [h_psi_eq] at h_set

rw [h_set] at h_subst

have h_bot' : substHenkinForm c (ArithTerm.var (s := s) v) (ArithForm.bottom (s := s)) = ArithForm.bottom (s := s) := rfl

rw [h_bot'] at h_subst

have h_self : substForm v (ArithTerm.var v) ψ = ψ := substForm_self_var ψ v

rw [h_self] at h_subst

have h_neg_H := SPA_Deduction_Internal.neg_intro Γ (ArithForm.impl (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) ψ) h_subst

have h_neg_exists := imp_neg_elim_left Γ _ _ h_neg_H

have h_neg_psi := imp_neg_elim_right Γ _ _ h_neg_H

have h_forall := SPA_Deduction_Internal.forall_intro Γ v (ArithForm.neg ψ) h_neg_psi h_fresh_w_Gamma

have h_bot_final := SPA_Deduction_Internal.neg_elim Γ (ArithForm.forall_form v (ArithForm.neg ψ)) h_forall h_neg_exists

exact h_con h_bot_final

Soporte generalizado para cotas y frescura

Se presentan tres resultados que generalizan la maquinaria de cotas y frescura a conjuntos finitos de fórmulas y a derivaciones arbitrarias, culminando con un teorema que garantiza la existencia de variables frescas respecto a una derivación y un conjunto de hipótesis.

Lema `NotFreeForm_of_NotOccurForm`: No ocurrencia implica no libertad

Versión generalizada (para cualquier estrato s) del lema que establece que si una variable no ocurre en absoluto en una fórmula, entonces no es libre en ella:

$$ \forall s \in \mathbb{N},\ \forall v \in \mathbb{N},\ \forall \varphi \in \text{ArithForm } s,\ \text{NotOccurForm } v\ \varphi \implies \text{NotFreeForm } v\ \varphi $$

La demostración es idéntica a la versión para $s = 0$ presentada anteriormente, pero sin restricción de estrato. El caso `forall_form n f` distingue si $v = n$ (ligada, por tanto no libre) o $v \neq n$ (se aplica la hipótesis de inducción).

Lema `max_var_list_ge_general`: Cota para fórmulas en listas (versión generalizada)

Generaliza `max_var_list_ge` a cualquier estrato s:

$$ \forall s \in \mathbb{N},\ \forall L \in \text{List}(\text{ArithForm } s),\ \forall \varphi \in L,\ \text{max_var_form}(\varphi) \le \text{max_var_list}(L) $$

La demostración es idéntica al caso $s = 0$, por inducción sobre la lista.

Definición `max_var_set`: Cota para conjuntos finitos

Extiende la noción de máximo de variables a conjuntos finitos de fórmulas (representados como `Finset`), utilizando la conversión a lista:

$$ \text{max_var_set}(\Delta) := \text{max_var_list}(\Delta.\text{toList}) $$

La definición se marca como `noncomputable` porque depende de `Finset.toList`, que no tiene un orden canónico.

Lema `max_var_form_le_max_var_set`: Cota para elementos de un conjunto finito

Establece que toda fórmula en un conjunto finito tiene su máximo de variables acotado por el máximo del conjunto:

$$ \forall \psi \in \Delta,\ \text{max_var_form}(\psi) \le \text{max_var_set}(\Delta) $$

La demostración utiliza `Finset.mem_toList` para convertir la pertenencia al conjunto en pertenencia a la lista, y luego aplica `max_var_list_ge_general`.

Teorema `exists_fresh_var_of_deriv`: Existencia de variable fresca para una derivación

Establece un resultado fundamental de carácter metacompacto: para cualquier derivación y cualquier conjunto de hipótesis acotado, existe una variable fresca que no aparece en la derivación ni es libre en las hipótesis:

$$ \forall d : \Gamma \vdash \varphi,\ \forall M \in \mathbb{N},\ (\forall \psi \in \Gamma,\ \text{max_var_form}(\psi) \le M) \implies \exists z > M,\ \text{FreshForDerivation } z\ d \land \text{NotFreeInSet } z\ \Gamma $$

Estructura de la demostración

1. Cota de la derivación: Por `exists_max_var_deriv`, existe m tal que `MaxVarDeriv h m`.

2. Construcción de la cota unificada: Sea $M' := \max(m, M)$ y $z := M' + 1$. Entonces $z > m$ y $z > M$.

3. Frescura para la derivación: Por `FreshForDerivation_of_gt_max_var`, dado que $z > m$, se tiene `FreshForDerivation z h`.

4. Frescura para las hipótesis: Para cada $\psi \in \Gamma$, se tiene $\text{max_var_form}(\psi) \le M < z$. Por `NotOccurForm_of_gt_max_var`, z no ocurre en ψ, y por `NotFreeForm_of_NotOccurForm`, z no es libre en ψ. Así, `NotFreeInSet z Γ`.

Significado

`exists_fresh_var_of_deriv` es un resultado de gran importancia metodológica. Garantiza que siempre es posible elegir una variable completamente nueva respecto a una derivación dada y a sus hipótesis. Esta propiedad de "frescura global" es la que

permite aplicar las reglas de generalización universal y los lemas de cambio de variable ligada de manera irrestricta.

En la práctica, cada vez que se necesita una variable fresca (por ejemplo, para la regla `forall_intro`, para la construcción de testigos de Henkin, o para la eliminación de constantes de Henkin), este teorema asegura que existe una variable suficientemente grande que satisface todas las condiciones de frescura necesarias simultáneamente. La cota M puede tomarse como el máximo de las variables en Γ (o 0 si Γ es vacío), y el teorema devuelve una variable $z > M$ con las propiedades deseadas.

Este resultado cierra la brecha entre las construcciones locales (donde se necesitan variables frescas para pasos individuales) y la construcción global de la secuencia de Henkin-Lindenbaum, donde se necesita una provisión infinita de variables y constantes frescas a lo largo de una enumeración completa del lenguaje. La combinación de `exists_max_var_deriv` (existencia de cota), `FreshForDerivation_of_gt_max_var` (frescura por cota superior) y `exists_fresh_var_of_deriv` (existencia de variable fresca) proporciona un mecanismo robusto y uniforme para la generación de símbolos frescos en cualquier contexto deductivo.

```
--
============================================================

-- SOPORTE GENERALIZADO PARA COTAS Y FRESCURA (CORREGIDO)

--
============================================================


lemma NotFreeForm_of_NotOccurForm {s : Nat} (v : Nat) (φ : ArithForm s) (h :
NotOccurForm v φ) : NotFreeForm v φ := by
```

```
  induction φ generalizing v with
  | bottom => trivial
  | eq t1 t2 => exact h
  | impl f1 f2 ih1 ih2 =>
    exact ⟨ih1 v h.1, ih2 v h.2⟩
  | neg f ih =>
    exact ih v h
  | forall_form n f ih =>
    dsimp [NotFreeForm]
    by_cases h_vn : v = n
    · rw [if_pos h_vn]
      exact True.intro
    · rw [if_neg h_vn]
      exact ih v h.2
  | provable_op _ _ => exact h
  | true_op _ _ => exact h
  | list_nth_eq _ _ _ => exact h

lemma max_var_list_ge_general {s : Nat} (L : List (ArithForm s)) (φ : ArithForm s) (h :
φ ∈ L) :
    max_var_form φ ≤ max_var_list L := by
  induction L with
  | nil => contradiction
  | cons ψ L' ih =>
    unfold max_var_list
    cases h
    · exact Nat.le_max_left _ _
    · rename_i h'
```

```
    exact Nat.le_trans (ih h') (Nat.le_max_right _ _)

/-- Marcado como noncomputable debido a la dependencia con Finset.toList -/
noncomputable def max_var_set {s : Nat} (Δ : Finset (ArithForm s)) : Nat :=
  max_var_list Δ.toList

lemma max_var_form_le_max_var_set {s : Nat} (Δ : Finset (ArithForm s)) (ψ : ArithForm s) (h : ψ ∈ Δ) :
    max_var_form ψ ≤ max_var_set Δ := by
  dsimp [max_var_set]
  apply max_var_list_ge_general
  rw [Finset.mem_toList]
  exact h

theorem exists_fresh_var_of_deriv {s : Nat} {Γ : Set (ArithForm s)} {φ : ArithForm s}
    (h : SPA_Deduction_Internal s Γ φ) (M : Nat)
    (h_Γ : ∀ ψ ∈ Γ, max_var_form ψ ≤ M)
    (_h_φ : max_var_form φ ≤ M) :
    ∃ z, z > M ∧ FreshForDerivation z h ∧ NotFreeInSet z Γ := by
  rcases exists_max_var_deriv h with ⟨m, hm⟩
  let M' := Nat.max m M
  let z := M' + 1
  use z
  have h_ge_m : m ≤ M' := Nat.le_max_left m M
  have h_ge_M : M ≤ M' := Nat.le_max_right m M
  have hz_gt_m : z > m := by omega
  have hz_gt_M : z > M := by omega
  refine ⟨hz_gt_M, ?_, ?_⟩
```

· exact FreshForDerivation_of_gt_max_var h m hm z hz_gt_m

· intro ψ hψ

have h_bound := h_Γ ψ hψ

have h_gt : z > max_var_form ψ := by omega

have h_not_occur := NotOccurForm_of_gt_max_var ψ z h_gt

exact NotFreeForm_of_NotOccurForm z ψ h_not_occur

Lemas auxiliares de constantes de Henkin: Invarianza bajo sustitución de variables

Se demuestran dos lemas que establecen una propiedad fundamental de las cotas de constantes de Henkin: la sustitución de una variable por otra variable no modifica el máximo índice de constante de Henkin en un término o fórmula.

Lema `maxHenkinTerm_substTerm_var`: Invarianza en términos

Establece que sustituir una variable v por otra variable w no altera el máximo índice de constante de Henkin en un término:

$$ \forall t \in \text{ArithTerm } s,\ \forall v, w \in \mathbb{N},\ \text{maxHenkinTerm}(\text{substTerm } v\ (\text{var } w)\ t) = \text{maxHenkinTerm}(t) $$

Estructura de la demostración

Por inducción sobre t:

- **`var n`:** La sustitución reemplaza la variable por otra variable (si $n = v$) o la deja igual. En ambos casos, `maxHenkinTerm` es 0 (las variables no contienen constantes de Henkin).

- **`henkin c`:** La sustitución de variables no afecta a las constantes de Henkin, por lo que el máximo índice se preserva.

- **Casos recursivos:** La sustitución se propaga a los subtérminos, y `maxHenkinTerm` se calcula como el máximo de los subtérminos. Las hipótesis de inducción garantizan la igualdad para cada subtérmino.

Intuición

La propiedad es estructuralmente evidente: la sustitución de una variable por otra solo afecta a los nodos `var` del árbol sintáctico, mientras que `maxHenkinTerm` solo depende de los nodos `henkin`. Como estas dos clases de nodos son disjuntas, la operación de sustitución de variables no tiene efecto sobre la cota de constantes de Henkin.

Lema `maxHenkinForm_substForm_var`: Invarianza en fórmulas

Extiende el resultado anterior al nivel de fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \text{maxHenkinForm}(\text{substForm } v\ (\text{var } w)\ \varphi) = \text{maxHenkinForm}(\varphi) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos con términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`):** Se aplica `maxHenkinTerm_substTerm_var` a cada subtérmino.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción a las subfórmulas.

- **`forall_form n f`:** Si la sustitución no penetra ($v = n$), la fórmula no cambia. Si penetra, se aplica la hipótesis de inducción al cuerpo f.

Significado

Estos lemas, aunque técnicamente simples, son importantes para la construcción de Henkin porque garantizan que la frescura de las constantes de Henkin no se ve comprometida por las operaciones de sustitución de variables que ocurren durante la construcción de los axiomas de testigo. En particular, cuando se construye `freshEx` como $\neg \forall w\ \neg (\psi[v := \text{var } w])$, la sustitución $[v := \text{var } w]$ no introduce nuevas constantes de Henkin: `maxHenkinForm(freshEx) = maxHenkinForm(ψ)`. Esto asegura que la constante de Henkin fresca c (elegida con índice mayor que `maxHenkinForm(freshEx)`) es efectivamente nueva respecto a `freshEx`.

La independencia entre las operaciones sobre variables y sobre constantes de Henkin refleja la ortogonalidad del diseño sintáctico del sistema SPA: las variables y las constantes de Henkin pertenecen a espacios de nombres disjuntos, y las operaciones sobre unos no afectan a los otros. Esta separación es la que permite el tratamiento modular de la frescura: la frescura de variables se controla mediante `max_var_*`, y la frescura de constantes de Henkin mediante `maxHenkin*`, sin interferencia mutua.

```
--
================================================================
-- FASE 10: TEOREMA DE CONSISTENCIA DE LA SECUENCIA DE HENKIN
--
================================================================


--///////////////    Lemas Auxiliares de Constantes de Henkin

lemma maxHenkinTerm_substTerm_var {s : Nat} (t : ArithTerm s) (v : Nat) (w : Nat) :
    maxHenkinTerm (substTerm v (ArithTerm.var w) t) = maxHenkinTerm t := by
  induction t with
  | var n =>
    unfold substTerm
    split_ifs <;> rfl
  | const c => rfl
  | henkin c => rfl
  | zero => rfl
  | succ t' ih => exact ih
  | add t1 t2 ih1 ih2 => dsimp [substTerm, maxHenkinTerm]; rw [ih1, ih2]
  | mul t1 t2 ih1 ih2 => dsimp [substTerm, maxHenkinTerm]; rw [ih1, ih2]
  | exp t1 t2 ih1 ih2 => dsimp [substTerm, maxHenkinTerm]; rw [ih1, ih2]
  | pred t' ih => exact ih

lemma maxHenkinForm_substForm_var {s : Nat} (φ : ArithForm s) (v : Nat) (w : Nat) :
    maxHenkinForm (substForm v (ArithTerm.var w) φ) = maxHenkinForm φ := by
  induction φ generalizing v with
```

```
  | bottom => rfl

  | eq t1 t2 =>

    change maxHenkinForm (ArithForm.eq (substTerm v (ArithTerm.var w) t1) (substTerm v (ArithTerm.var w) t2)) = maxHenkinForm (ArithForm.eq t1 t2)

    dsimp [maxHenkinForm]

    rw [maxHenkinTerm_substTerm_var t1 v w, maxHenkinTerm_substTerm_var t2 v w]

  | impl f1 f2 ih1 ih2 =>

    change Nat.max (maxHenkinForm (substForm v (ArithTerm.var w) f1)) (maxHenkinForm (substForm v (ArithTerm.var w) f2)) = Nat.max (maxHenkinForm f1) (maxHenkinForm f2)

    rw [ih1 v, ih2 v]

  | neg f ih =>

    change maxHenkinForm (substForm v (ArithTerm.var w) f) = maxHenkinForm f

    rw [ih v]

  | forall_form n f ih =>

    dsimp [substForm]

    split_ifs with h_vn

    · rfl

    · change maxHenkinForm (substForm v (ArithTerm.var w) f) = maxHenkinForm f

      rw [ih v]

  | provable_op heq t =>

    change maxHenkinTerm (substTerm v (ArithTerm.var w) t) = maxHenkinTerm t

    rw [maxHenkinTerm_substTerm_var t v w]

  | true_op heq t =>

    change maxHenkinTerm (substTerm v (ArithTerm.var w) t) = maxHenkinTerm t

    rw [maxHenkinTerm_substTerm_var t v w]

  | list_nth_eq t1 t2 t3 =>

    change Nat.max (maxHenkinTerm (substTerm v (ArithTerm.var w) t1)) (Nat.max (maxHenkinTerm (substTerm v (ArithTerm.var w) t2)) (maxHenkinTerm (substTerm
```

v (ArithTerm.var w) t3))) = Nat.max (maxHenkinTerm t1) (Nat.max (maxHenkinTerm t2) (maxHenkinTerm t3))

rw [maxHenkinTerm_substTerm_var t1 v w, maxHenkinTerm_substTerm_var t2 v w, maxHenkinTerm_substTerm_var t3 v w]

Lemas `substTerm_substTerm_henkin` y `substForm_substForm_henkin`

Se demuestran dos lemas que establecen una propiedad de cancelación para sustituciones que involucran constantes de Henkin y variables frescas. Estos resultados son esenciales para la manipulación de los axiomas de testigo durante la construcción de Henkin.

Lema `substTerm_substTerm_henkin`: Cancelación de sustituciones con variable fresca y constante

Establece que si una variable w no ocurre en un término t, entonces sustituir primero v por $\text{var } w$ y luego w por $\text{henkin } c$ es equivalente a sustituir directamente v por $\text{henkin } c$:

$$ \forall t \in \text{ArithTerm } s,\ \forall v, w \in \mathbb{N},\ \forall c \in \text{HenkinConst}, $$

$$ \text{NotOccur } w\ t \implies \text{substTerm } w\ (\text{henkin } c)\ (\text{substTerm } v\ (\text{var } w)\ t) = \text{substTerm } v\ (\text{henkin } c)\ t $$

Estructura de la demostración

Por inducción sobre t:

- **`var n`:** Se analiza si $n = v$:

  - Si $n = v$, la primera sustitución produce $\text{var } w$, y la segunda la reemplaza por $\text{henkin } c$. El lado derecho sustituye directamente v por $\text{henkin } c$, produciendo el mismo resultado.

  - Si $n \neq v$, la primera sustitución no afecta. La segunda sustitución solo afectaría si $n = w$, pero `NotOccur w (var n)` fuerza $w \neq n$. Ambos lados preservan $\text{var } n$.

- **`henkin c'`:** Las sustituciones de variables no afectan a las constantes de Henkin. Ambos lados preservan $\text{henkin } c'$.

- **Casos recursivos:** Se invierte la hipótesis `NotOccur w t` y se aplican las hipótesis de inducción.

Intuición

Este lema captura la esencia de la construcción de testigos de Henkin. La fórmula `freshEx` se construye como $\neg \forall w\ \neg (\psi[v := \text{var } w])$, y el axioma de testigo es `freshEx → ψ[v := henkin c]`. La equivalencia entre sustituir v por $\text{var } w$ y luego w por $\text{henkin } c$, versus sustituir directamente v por $\text{henkin } c$, es lo que permite conectar `freshEx` (que usa $\text{var } w$) con el consecuente del axioma (que usa $\text{henkin } c$).

Lema `substForm_substForm_henkin`: Cancelación en fórmulas

Extiende el resultado anterior al nivel de fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \forall c \in \text{HenkinConst}, $$

$$ \text{NotOccurForm } w\ \varphi \implies \text{substForm } w\ (\text{henkin } c)\ (\text{substForm } v\ (\text{var } w)\ \varphi) = \text{substForm } v\ (\text{henkin } c)\ \varphi $$

Estructura de la demostración

La prueba procede por casos sobre la estructura de φ (usando `match` en lugar de inducción), aplicando `substTerm_substTerm_henkin` a los subtérminos y las hipótesis de inducción a las subfórmulas. El caso `forall_form n f` distingue si $v = n$ (la sustitución no penetra) o $v \neq n$. En el subcaso $v \neq n$, la condición $w \neq n$ (provista por `NotOccurForm w (forall_form n f)`) garantiza que la sustitución sobre w también atraviesa el cuantificador, y la hipótesis de inducción se aplica a f.

Significado

Estos lemas de cancelación cierran el círculo de la construcción de Henkin. Muestran que la introducción de la variable fresca w en `freshEx` y su posterior reemplazo por la constante de Henkin c en el consecuente del axioma son operaciones inversas (módulo la sustitución original $[v := \text{henkin } c]$). Esta propiedad es la que garantiza que el axioma de testigo es correcto: la existencia de un testigo para ψ (expresada como `freshEx`) implica la existencia de una constante de Henkin que satisface ψ (expresada como $\psi[v := \text{henkin } c]$).

La hipótesis `NotOccurForm w φ` es crucial: garantiza que la variable fresca w no interfiere con otras partes de la fórmula. Esta condición se satisface automáticamente en la construcción porque w se elige como una variable completamente nueva (mayor que `max_var_form` de todas las fórmulas involucradas). La combinación de `substForm_substForm_henkin` con `substForm_substForm_self` (cancelación de sustituciones intercambiadas) y `substForm_self_var` (identidad de la sustitución de

una variable por sí misma) proporciona un álgebra completa de sustituciones que permite manipular fórmulas con variables y constantes de Henkin de manera sistemática y segura.

```
lemma substTerm_substTerm_henkin {s : Nat} (t : ArithTerm s) (v w : Nat) (c : HenkinConst) (hw : NotOccur w t) :
  substTerm w (ArithTerm.henkin c) (substTerm v (ArithTerm.var w) t) = substTerm v (ArithTerm.henkin c) t := by
 induction t with
 | var n =>
  simp only [substTerm_var]
  split_ifs with h1
  · subst h1
   simp only [substTerm_var]
   split_ifs
   · rfl
   · rfl
  · simp only [substTerm_var]
   split_ifs with h2
   · cases hw with | var _ h_ne => contradiction
   · rfl
 | const c' => rfl
 | henkin c' => rfl
 | zero => rfl
 | succ t' ih =>
```

```
    cases hw with | succ _ h_sub =>
    dsimp [substTerm]
    rw [ih h_sub]
  | add t1 t2 ih1 ih2 =>
    cases hw with | add _ _ h1 h2 =>
    dsimp [substTerm]
    rw [ih1 h1, ih2 h2]
  | mul t1 t2 ih1 ih2 =>
    cases hw with | mul _ _ h1 h2 =>
    dsimp [substTerm]
    rw [ih1 h1, ih2 h2]
  | exp t1 t2 ih1 ih2 =>
    cases hw with | exp _ _ h1 h2 =>
    dsimp [substTerm]
    rw [ih1 h1, ih2 h2]
  | pred t' ih =>
    cases hw with | pred _ h_sub =>
    dsimp [substTerm]
    rw [ih h_sub]

lemma substForm_substForm_henkin {s : Nat} (φ : ArithForm s) (v w : Nat) (c :
HenkinConst) (hw : NotOccurForm w φ) :
    substForm w (ArithTerm.henkin c) (substForm v (ArithTerm.var w) φ) = substForm
v (ArithTerm.henkin c) φ :=
  match φ with
  | ArithForm.bottom => rfl
  | ArithForm.eq t1 t2 => by
    have hw_left : NotOccur w t1 := hw.left
    have hw_right : NotOccur w t2 := hw.right
```

```
    rw [substForm_eq, substForm_eq]
    rw [substTerm_substTerm_henkin t1 v w c hw_left, substTerm_substTerm_henkin t2 v w c hw_right]
    rfl
  | ArithForm.impl f1 f2 => by
    have hw_left : NotOccurForm w f1 := hw.left
    have hw_right : NotOccurForm w f2 := hw.right
    rw [substForm_impl, substForm_impl]
    rw [substForm_substForm_henkin f1 v w c hw_left, substForm_substForm_henkin f2 v w c hw_right]
    rfl
  | ArithForm.neg f => by
    have hw_inner : NotOccurForm w f := hw
    rw [substForm_neg, substForm_neg]
    rw [substForm_substForm_henkin f v w c hw_inner]
    rfl
  | ArithForm.forall_form n f => by
    have hw_left : w ≠ n := hw.left
    have hw_right : NotOccurForm w f := hw.right
    by_cases h_v : v = n
    · subst h_v
      rw [substForm_forall_form, if_pos rfl]
      rw [substForm_forall_form, if_neg hw_left]
      rw [substForm_fresh f w (ArithTerm.henkin c) hw_right]
      rw [substForm_forall_form, if_pos rfl]
    · rw [substForm_forall_form, if_neg h_v]
      rw [substForm_forall_form, if_neg hw_left]
      rw [substForm_substForm_henkin f v w c hw_right]
      rw [substForm_forall_form, if_neg h_v]
```

```
  | ArithForm.provable_op heq t => by
    have hw_inner : NotOccur w t := hw
    rw [substForm_provable_op, substForm_provable_op]
    rw [substTerm_substTerm_henkin t v w c hw_inner]
    rfl
  | ArithForm.true_op heq t => by
    have hw_inner : NotOccur w t := hw
    rw [substForm_true_op, substForm_true_op]
    rw [substTerm_substTerm_henkin t v w c hw_inner]
    rfl
  | ArithForm.list_nth_eq t1 t2 t3 => by
    have hw_t1 : NotOccur w t1 := hw.left
    have hw_t2 : NotOccur w t2 := hw.right.left
    have hw_t3 : NotOccur w t3 := hw.right.right
    rw [substForm_list_nth_eq, substForm_list_nth_eq]
    rw [substTerm_substTerm_henkin t1 v w c hw_t1, substTerm_substTerm_henkin t2
v w c hw_t2, substTerm_substTerm_henkin t3 v w c hw_t3]
    rfl
```

Lemas de preservación de `ConstantNotOccur` bajo sustitución de variables

Se demuestran dos lemas que establecen que la propiedad de que una constante de Henkin no ocurre en un término o fórmula se preserva bajo sustituciones de variables por otras variables.

Lema `ConstantNotOccur_substTerm_var`: Preservación en términos

Establece que si una constante de Henkin c no ocurre en un término t, entonces c tampoco ocurre en el resultado de sustituir cualquier variable v por cualquier variable w en t:

$$ \forall t \in \text{ArithTerm } s,\ \forall c \in \text{HenkinConst},\ \forall v, w \in \mathbb{N},\ \text{ConstantNotOccur } c\ t \implies \text{ConstantNotOccur } c\ (\text{substTerm } v\ (\text{var } w)\ t) $$

Estructura de la demostración

Por inducción sobre t:

- **`var n`:** Si la sustitución se aplica ($n = v$), el resultado es $\text{var } w$, que no contiene constantes de Henkin. Si no se aplica, t permanece inalterado y la hipótesis se preserva.

- **`henkin c'`:** La sustitución de variables no afecta a las constantes de Henkin, por lo que la propiedad se preserva directamente.

- **Casos recursivos:** Se invierte la hipótesis `ConstantNotOccur c t` y se aplican las hipótesis de inducción a los subtérminos.

Intuición

La sustitución de una variable por otra solo puede introducir nuevas variables, no nuevas constantes de Henkin. Por tanto, si una constante no aparecía originalmente, sigue sin aparecer tras la sustitución.

Lema `ConstantNotOccurForm_substForm_var`: Preservación en fórmulas

Extiende el resultado anterior al nivel de fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall c \in \text{HenkinConst},\ \forall v, w \in \mathbb{N},\ \text{ConstantNotOccurForm } c\ \varphi \implies \text{ConstantNotOccurForm } c\ (\text{substForm } v\ (\text{var } w)\ \varphi) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos con términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`):** Se aplica `ConstantNotOccur_substTerm_var` a cada subtérmino.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción a las subfórmulas.

- **`forall_form n f`:** Si $v = n$, la sustitución no penetra y la fórmula no cambia, preservando la propiedad. Si $v \neq n$, la sustitución atraviesa el cuantificador y se aplica la hipótesis de inducción al cuerpo f. La condición `ConstantNotOccurForm c (forall_form n f)` proporciona `ConstantNotOccurForm c f`, que es exactamente lo necesario para la hipótesis de inducción.

Significado

Estos lemas son fundamentales para la construcción de Henkin porque garantizan que la frescura de las constantes de Henkin es estable bajo las operaciones de renombramiento de variables que ocurren durante la construcción de `freshEx`. En particular, cuando se construye `freshEx` como $\neg \forall w\ \neg (\psi[v := \text{var } w])$, si ψ no contiene una constante de Henkin c, entonces `freshEx` tampoco la contiene. Esto asegura que la nueva constante de Henkin generada para ser testigo de esta fórmula existencial es efectivamente fresca respecto a `freshEx`.

La preservación de `ConstantNotOccurForm` bajo sustituciones de variables es una propiedad de monotonía: las operaciones sobre variables no introducen constantes de Henkin. Esta separación entre los dos espacios de nombres (variables y constantes de Henkin) es una característica del diseño sintáctico que simplifica considerablemente el razonamiento sobre frescura, ya que permite tratar la frescura de variables y la frescura de constantes de manera independiente. Los lemas de esta sección formalizan esta independencia.

```
lemma ConstantNotOccur_substTerm_var {s : Nat} (t : ArithTerm s) (c : HenkinConst)
(v w : Nat) (hc : ConstantNotOccur c t) :
  ConstantNotOccur c (substTerm v (ArithTerm.var w) t) := by
 induction t with
 | var n =>
  unfold substTerm
  split_ifs with h
  · exact ConstantNotOccur.var w
  · exact hc
 | const c' => exact ConstantNotOccur.const c'
```

```
 | henkin c' =>
  unfold substTerm
  exact hc
 | zero => exact ConstantNotOccur.zero
 | succ t' ih =>
  cases hc with | succ _ h_sub =>
  unfold substTerm; exact ConstantNotOccur.succ _ (ih h_sub)
 | add t1 t2 ih1 ih2 =>
  cases hc with | add _ _ h1 h2 =>
  unfold substTerm; exact ConstantNotOccur.add _ _ (ih1 h1) (ih2 h2)
 | mul t1 t2 ih1 ih2 =>
  cases hc with | mul _ _ h1 h2 =>
  unfold substTerm; exact ConstantNotOccur.mul _ _ (ih1 h1) (ih2 h2)
 | exp t1 t2 ih1 ih2 =>
  cases hc with | exp _ _ h1 h2 =>
  unfold substTerm; exact ConstantNotOccur.exp _ _ (ih1 h1) (ih2 h2)
 | pred t' ih =>
  cases hc with | pred _ h_sub =>
  unfold substTerm; exact ConstantNotOccur.pred _ (ih h_sub)

lemma ConstantNotOccurForm_substForm_var {s : Nat} (φ : ArithForm s) (c :
HenkinConst) (v w : Nat) (hc : ConstantNotOccurForm c φ) :
  ConstantNotOccurForm c (substForm v (ArithTerm.var w) φ) := by
 induction φ generalizing v with
 | bottom => exact ConstantNotOccurForm.bottom
 | eq t1 t2 =>
  cases hc with | eq _ _ h1 h2 =>
  dsimp [substForm]
```

```
    exact ConstantNotOccurForm.eq _ _ (ConstantNotOccur_substTerm_var t1 c v w h1)
(ConstantNotOccur_substTerm_var t2 c v w h2)
  | impl f1 f2 ih1 ih2 =>
    cases hc with | impl _ _ h1 h2 =>
    dsimp [substForm]
    exact ConstantNotOccurForm.impl _ _ (ih1 v h1) (ih2 v h2)
  | neg f ih =>
    cases hc with | neg _ h' =>
    dsimp [substForm]
    exact ConstantNotOccurForm.neg _ (ih v h')
  | forall_form n f ih =>
    cases hc with | forall_form _ _ h' =>
    dsimp [substForm]
    split_ifs with h_vn
    · exact ConstantNotOccurForm.forall_form n f h'
    · exact ConstantNotOccurForm.forall_form n _ (ih v h')
  | provable_op heq t =>
    cases hc with | provable_op _ _ h' =>
    dsimp [substForm]
    exact ConstantNotOccurForm.provable_op heq _ (ConstantNotOccur_substTerm_var
t c v w h')
  | true_op heq t =>
    cases hc with | true_op _ _ h' =>
    dsimp [substForm]
    exact ConstantNotOccurForm.true_op heq _ (ConstantNotOccur_substTerm_var t c v
w h')
  | list_nth_eq t1 t2 t3 =>
    cases hc with | list_nth_eq _ _ _ h1 h2 h3 =>
    dsimp [substForm]
```

```
  exact ConstantNotOccurForm.list_nth_eq _ _ _ (ConstantNotOccur_substTerm_var t1 c v w h1) (ConstantNotOccur_substTerm_var t2 c v w h2) (ConstantNotOccur_substTerm_var t3 c v w h3)
```

Lema `SPA_Deduction_Internal_finite_support`: Soporte finito de las derivaciones

Se demuestra un teorema estructural fundamental sobre el sistema deductivo: toda derivación utiliza solo un número finito de hipótesis. Formalmente, si $\Gamma \vdash \varphi$, entonces existe un subconjunto finito $\Gamma_0 \subseteq \Gamma$ tal que $\Gamma_0 \vdash \varphi$:

$$ \forall \Gamma \subseteq \text{ArithForm } s,\ \forall \varphi \in \text{ArithForm } s,\ \Gamma \vdash \varphi \implies \exists \Gamma_0 \subseteq_{\text{fin}} \Gamma,\ \Gamma_0 \vdash \varphi $$

Estructura de la demostración

La prueba procede por inducción sobre la derivación h, cubriendo todas las reglas de inferencia. Para cada regla, se construye un conjunto finito Γ_0 a partir de los conjuntos finitos de las subderivaciones (obtenidos por hipótesis de inducción):

- **Reglas sin premisas (`hyp`, `eq_refl`, `ax_peano1`, `ax_peano2`):** El soporte finito es simplemente la fórmula involucrada (para `hyp`) o el conjunto vacío (para los axiomas, que no dependen de hipótesis).

- **Reglas unarias (`double_neg`, `neg`, `eq_symm`, `eq_succ`, `eq_pred`, `weaken`, `forall_elim`):** El soporte finito se hereda directamente de la subderivación.

- **Reglas que descargan hipótesis (`imp_intro`, `neg_intro`):** La construcción es más delicada. La subderivación puede usar la fórmula φ que se va a descargar. El soporte finito de la subderivación podría contener φ; al descargarla, se debe eliminar del conjunto de hipótesis. Por eso se toma $\Gamma_0' \setminus \{\varphi\}$ (implementado como `Γ0'.erase φ`). Se verifica que este conjunto está contenido en Γ (porque la fórmula descargada estaba en Γ o era la propia φ, que ya no está). Luego se reconstruye la derivación usando `weaken` para adaptar el contexto.

- **Reglas binarias (`imp_elim`, `neg_elim`, `eq_trans`, `eq_add`, `eq_mul`, `eq_exp`, `eq_provable_op`, `eq_true_op`, `eq_list_nth_eq_*`, `eq_subst`, `induction`):** Se toma la unión de los soportes finitos de ambas subderivaciones. Se utiliza `weaken` para llevar cada subderivación al contexto unificado, y luego se aplica la regla correspondiente.

Significado

`SPA_Deduction_Internal_finite_support` es la propiedad de compacidad sintáctica del sistema deductivo: una fórmula es demostrable a partir de un conjunto de hipótesis si y solo si es demostrable a partir de un subconjunto finito de ellas. Esta propiedad es fundamental para la lógica de primer orden y es utilizada profusamente en las demostraciones de consistencia de los límites de Henkin y Lindenbaum.

En particular, es el lema que permite pasar de "el límite es consistente porque cada paso finito lo es" a "el límite es consistente": si el límite fuera inconsistente, existiría una derivación de $\bot$ a partir de él. Por soporte finito, esa derivación usaría solo un número finito de hipótesis del límite. Pero un conjunto finito de hipótesis del límite está contenido en algún paso finito de la secuencia (por monotonía). Si cada paso finito es consistente, el límite también lo es. Este argumento de compacidad es la

columna vertebral de las demostraciones de `GammaLimit1_consistent` y `LindenbaumLimit1_consistent`.

La demostración del lema es extensa pero mecánica: cada regla de inferencia requiere un tratamiento específico para construir el soporte finito, reflejando la estructura de la regla. Los casos de descarga de hipótesis (`imp_intro`, `neg_intro`) son los más técnicos porque requieren eliminar la fórmula descargada del soporte, lo cual se implementa mediante la operación `Finset.erase`.

```
--
============================================================

-- SOPORTE FINITO Y CONSISTENCIA DEL LÍMITE DE HENKIN

--
============================================================


lemma SPA_Deduction_Internal_finite_support {s} {Γ : Set (ArithForm s)} {φ} (h :
SPA_Deduction_Internal s Γ φ) :

   ∃ (Γ0 : Finset (ArithForm s)), (Γ0 : Set (ArithForm s)) ⊆ Γ ∧ SPA_Deduction_Internal
s (Γ0 : Set (ArithForm s)) φ := by

  classical

  induction h with

  | hyp Γ φ h_in =>

    use {φ}

    constructor
```

```
  · intro x hx
    have h_mem := Finset.mem_coe.mp hx
    rw [Finset.mem_singleton] at h_mem
    subst h_mem
    exact h_in
  · apply SPA_Deduction_Internal.hyp
    exact Finset.mem_coe.mpr (Finset.mem_singleton_self φ)
| imp_intro Γ φ ψ h ih =>
  rcases ih with ⟨Γ0', h_sub, h_ded⟩
  use Γ0'.erase φ
  constructor
  · intro x hx
    have h_mem : x ∈ Γ0' := by
      simp only [Finset.mem_coe, Finset.mem_erase] at hx
      exact hx.2
    have h_in_insert := h_sub h_mem
    simp only [Set.mem_insert_iff] at h_in_insert
    cases h_in_insert with
    | inl h_eq =>
      exfalso
      simp only [Finset.mem_coe, Finset.mem_erase] at hx
      exact hx.1 h_eq
    | inr h_in => exact h_in
  · apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.weaken (Γ0' : Set (ArithForm s))
    · exact h_ded
    · intro x hx
      simp only [Set.mem_insert_iff]
```

```
    by_cases h_eq : x = φ
    · left; exact h_eq
    · right
      simp only [Finset.mem_coe, Finset.mem_erase]
      exact ⟨h_eq, hx⟩
| imp_elim Γ φ ψ h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.impl φ ψ) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) φ := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.imp_elim _ φ ψ h_ded1' h_ded2'
| double_neg Γ φ h ih =>
  rcases ih with ⟨Γ0, h_sub, h_ded⟩
  use Γ0
  refine ⟨h_sub, SPA_Deduction_Internal.double_neg _ _ h_ded⟩
| neg_intro Γ φ h ih =>
  rcases ih with ⟨Γ0', h_sub, h_ded⟩
```

```
  use Γ0'.erase φ
  constructor
  · intro x hx
    have h_mem : x ∈ Γ0' := by
      simp only [Finset.mem_coe, Finset.mem_erase] at hx
      exact hx.2
    have h_in_insert := h_sub h_mem
    simp only [Set.mem_insert_iff] at h_in_insert
    cases h_in_insert with
    | inl h_eq =>
      exfalso
      simp only [Finset.mem_coe, Finset.mem_erase] at hx
      exact hx.1 h_eq
    | inr h_in => exact h_in
  · apply SPA_Deduction_Internal.neg_intro
    apply SPA_Deduction_Internal.weaken (Γ0' : Set (ArithForm s))
    · exact h_ded
    · intro x hx
      simp only [Set.mem_insert_iff]
      by_cases h_eq : x = φ
      · left; exact h_eq
      · right
        simp only [Finset.mem_coe, Finset.mem_erase]
        exact ⟨h_eq, hx⟩
| neg_elim Γ φ h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
```

```
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) φ := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.neg φ) := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.neg_elim _ φ h_ded1' h_ded2'
| forall_intro Γ v φ h h_vΓ ih =>
  rcases ih with ⟨Γ0, h_sub, h_ded⟩
  use Γ0
  constructor
  · exact h_sub
  · apply SPA_Deduction_Internal.forall_intro _ v φ h_ded
    intro ψ hψ
    exact h_vΓ ψ (h_sub hψ)
| forall_elim Γ v φ t h h_free ih =>
  rcases ih with ⟨Γ0, h_sub, h_ded⟩
  use Γ0
  refine ⟨h_sub, SPA_Deduction_Internal.forall_elim _ v φ t h_ded h_free⟩
| eq_refl Γ t =>
  use ∅
  constructor
```

```
  · rw [Finset.coe_empty]
    exact Set.empty_subset _
  · exact SPA_Deduction_Internal.eq_refl _ t
| eq_symm Γ t1 t2 h ih =>
  rcases ih with ⟨Γ0, h_sub, h_ded⟩
  use Γ0
  refine ⟨h_sub, SPA_Deduction_Internal.eq_symm _ t1 t2 h_ded⟩
| eq_trans Γ t1 t2 t3 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t1 t2) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t2 t3) := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.eq_trans _ t1 t2 t3 h_ded1' h_ded2'
| eq_succ Γ t1 t2 h ih =>
  rcases ih with ⟨Γ0, h_sub, h_ded⟩
  use Γ0
  refine ⟨h_sub, SPA_Deduction_Internal.eq_succ _ t1 t2 h_ded⟩
```

```
| eq_add Γ t1 t2 u1 u2 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t1 t2) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq u1 u2) := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.eq_add _ t1 t2 u1 u2 h_ded1' h_ded2'
| eq_mul Γ t1 t2 u1 u2 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t1 t2) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
```

```
    · exact Set.subset_union_left
   have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq u1 u2) := by
    apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
    · exact h_ded2
    · exact Set.subset_union_right
   exact SPA_Deduction_Internal.eq_mul _ t1 t2 u1 u2 h_ded1' h_ded2'
| eq_exp Γ t1 t2 u1 u2 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t1 t2) := by
     apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
     · exact h_ded1
     · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq u1 u2) := by
     apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
     · exact h_ded2
     · exact Set.subset_union_right
    exact SPA_Deduction_Internal.eq_exp _ t1 t2 u1 u2 h_ded1' h_ded2'
| eq_pred Γ t1 t2 h ih =>
  rcases ih with ⟨Γ0, h_sub, h_ded⟩
  use Γ0
  refine ⟨h_sub, SPA_Deduction_Internal.eq_pred _ t1 t2 h_ded⟩
| eq_provable_op Γ h_eq t1 t2 h1 h2 ih1 ih2 =>
```

```
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t1 t2) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.provable_op h_eq
t1) := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.eq_provable_op _ h_eq t1 t2 h_ded1' h_ded2'
| eq_true_op Γ h_eq t1 t2 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t1 t2) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
```

```
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.true_op h_eq t1)
:= by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.eq_true_op _ h_eq t1 t2 h_ded1' h_ded2'
  | eq_list_nth_eq_1 Γ t1 t2 t3 u1 h1 h2 ih1 ih2 =>
   rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
   rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
   use Γ1 ∪ Γ2
   constructor
   · rw [Finset.coe_union]
     exact Set.union_subset h_sub1 h_sub2
   · rw [Finset.coe_union]
     have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t1 u1) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.list_nth_eq t1 t2
t3) := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.eq_list_nth_eq_1 _ t1 t2 t3 u1 h_ded1' h_ded2'
  | eq_list_nth_eq_2 Γ t1 t2 t3 u2 h1 h2 ih1 ih2 =>
   rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
   rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
   use Γ1 ∪ Γ2
```

```
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t2 u2) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.list_nth_eq t1 t2
t3) := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.eq_list_nth_eq_2 _ t1 t2 t3 u2 h_ded1' h_ded2'
 | eq_list_nth_eq_3 Γ t1 t2 t3 u3 h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t3 u3) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.list_nth_eq t1 t2
t3) := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
```

```
    · exact h_ded2
    · exact Set.subset_union_right
  exact SPA_Deduction_Internal.eq_list_nth_eq_3 _ t1 t2 t3 u3 h_ded1' h_ded2'
| eq_subst Γ v φ t1 t2 h1 h2 h_free1 h_free2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.eq t1 t2) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (substForm v t1 φ) := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.eq_subst _ v φ t1 t2 h_ded1' h_ded2' h_free1 h_free2
| weaken Γ φ Γ' h h_sub ih =>
  rcases ih with ⟨Γ0, h_sub0, h_ded⟩
  use Γ0
  refine ⟨Set.Subset.trans h_sub0 h_sub, h_ded⟩
| ax_peano1 Γ x =>
  use ∅
  constructor
  · rw [Finset.coe_empty]
```

```
    exact Set.empty_subset _
  · exact SPA_Deduction_Internal.ax_peano1 _ x
| ax_peano2 Γ x y =>
  use ∅
  constructor
  · rw [Finset.coe_empty]
    exact Set.empty_subset _
  · exact SPA_Deduction_Internal.ax_peano2 _ x y
| induction Γ φ v h1 h2 ih1 ih2 =>
  rcases ih1 with ⟨Γ1, h_sub1, h_ded1⟩
  rcases ih2 with ⟨Γ2, h_sub2, h_ded2⟩
  use Γ1 ∪ Γ2
  constructor
  · rw [Finset.coe_union]
    exact Set.union_subset h_sub1 h_sub2
  · rw [Finset.coe_union]
    have h_ded1' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (substForm v ArithTerm.zero
φ) := by
      apply SPA_Deduction_Internal.weaken (Γ1 : Set (ArithForm s))
      · exact h_ded1
      · exact Set.subset_union_left
    have h_ded2' : SPA_Deduction_Internal s (↑Γ1 ∪ ↑Γ2) (ArithForm.forall_form v
(ArithForm.impl φ (substForm v (ArithTerm.succ (ArithTerm.var v)) φ))) := by
      apply SPA_Deduction_Internal.weaken (Γ2 : Set (ArithForm s))
      · exact h_ded2
      · exact Set.subset_union_right
    exact SPA_Deduction_Internal.induction _ φ v h_ded1' h_ded2'
```

Lemas `GammaLimit_subset_GammaSeq` y `GammaLimit_consistent`

Se demuestran dos resultados fundamentales sobre el límite de la secuencia de Henkin: la reducción finitaria de la pertenencia al límite y la preservación de la consistencia en el límite.

Lema `GammaLimit_subset_GammaSeq`: Reducción finitaria del límite de Henkin

Establece que cualquier conjunto finito de fórmulas contenido en el límite de Henkin está ya contenido en algún paso finito de la secuencia:

$$ \forall \Gamma \subseteq \text{ArithForm } 0,\ \forall \Gamma_0 \subseteq_{\text{fin}} \text{GammaLimit}(\Gamma),\ \exists N \in \mathbb{N},\ \Gamma_0 \subseteq \text{GammaSeq}(\Gamma, N) $$

Estructura de la demostración

Por inducción sobre el conjunto finito Γ_0 (representado como `Finset`):

- **Caso vacío:** Trivial, tomando $N = 0$.

- **Caso inserción ($\Gamma_0 = \{\varphi\} \cup s_{\text{fin}}$):**

  1. De la hipótesis, $\varphi \in \text{GammaLimit}(\Gamma)$, por lo que existe n_φ con $\varphi \in \text{GammaSeq}(\Gamma, n_\varphi)$.

  2. El resto s_{fin} está contenido en el límite, y por hipótesis de inducción existe n_s con $s_{\text{fin}} \subseteq \text{GammaSeq}(\Gamma, n_s)$.

  3. Tomando $N = \max(n_\varphi, n_s)$, por monotonía de `GammaSeq` (`GammaSeq_mono_step`), ambas inclusiones se elevan al paso N, obteniendo $\Gamma_0 \subseteq \text{GammaSeq}(\Gamma, N)$.

Lema `GammaLimit_consistent`: Consistencia del límite de Henkin

Establece el resultado principal de la Fase 10: si todos los pasos finitos de la secuencia de Henkin son consistentes, entonces el límite también lo es:

$$ (\forall n \in \mathbb{N},\ \text{Consistent}(\text{GammaSeq } \Gamma\ n)) \implies \text{Consistent}(\text{GammaLimit } \Gamma) $$

Estructura de la demostración

Se asume que el límite es inconsistente, es decir, existe una derivación h_{bot} de $\bot$ a partir de $\text{GammaLimit}(\Gamma)$.

1. **Soporte finito:** Por `SPA_Deduction_Internal_finite_support`, existe un conjunto finito $\Gamma_0 \subseteq \text{GammaLimit}(\Gamma)$ tal que $\Gamma_0 \vdash \bot$.

2. **Reducción a un paso finito:** Por `GammaLimit_subset_GammaSeq`, existe N tal que $\Gamma_0 \subseteq \text{GammaSeq}(\Gamma, N)$.

3. **Debilitamiento:** Por la regla `weaken`, la derivación $\Gamma_0 \vdash \bot$ se extiende a $\text{GammaSeq}(\Gamma, N) \vdash \bot$.

4. **Contradicción:** Pero por hipótesis, $\text{GammaSeq}(\Gamma, N)$ es consistente, es decir, no existe tal derivación. Contradicción.

Significado

`GammaLimit_consistent` es uno de los teoremas centrales de la construcción de Henkin. Establece que el proceso de añadir axiomas de testigo para todas las fórmulas existenciales del lenguaje no introduce inconsistencias, siempre que cada paso individual preserve la consistencia. La demostración es una aplicación directa de la compacidad sintáctica (`SPA_Deduction_Internal_finite_support`) combinada con la monotonía de la secuencia de Henkin.

Este resultado, junto con `consistency_preservation_local` (que garantiza que cada paso individual preserva la consistencia), implica que si la teoría original Γ es consistente, entonces todo el límite de Henkin $\text{GammaLimit}(\Gamma)$ es consistente. Este es el paso fundamental en la demostración de completitud: a partir de una teoría consistente, se construye una extensión de Henkin consistente que está existencialmente saturada, sobre la cual se puede luego aplicar Lindenbaum para obtener una teoría maximal consistente con testigos de Henkin, y finalmente construir un modelo. La combinación de `consistency_preservation_local` (preservación local) y `GammaLimit_consistent` (preservación global) proporciona la garantía de que todo el proceso de extensión es seguro.

```
lemma GammaLimit_subset_GammaSeq (Γ : Set (ArithForm 0)) (Γ0 : Finset (ArithForm 0)) :

  ((Γ0 : Set (ArithForm 0)) ⊆ GammaLimit Γ) → ∃ N, (Γ0 : Set (ArithForm 0)) ⊆ GammaSeq Γ N := by

 classical

 induction Γ0 using Finset.induction_on with

 | empty =>

  intro _

  use 0

  rw [Finset.coe_empty]

  exact Set.empty_subset _

 | insert φ s_fin has ih =>

  intro h_sub

  have h_phi : φ ∈ GammaLimit Γ := h_sub (Finset.mem_coe.mpr (Finset.mem_insert_self φ s_fin))

  have h_s : (s_fin : Set (ArithForm 0)) ⊆ GammaLimit Γ := by

   intro x hx

   apply h_sub

   apply Finset.mem_coe.mpr

   exact Finset.mem_insert_of_mem (Finset.mem_coe.mp hx)

  rcases h_phi with ⟨n_phi, hn_phi⟩

  rcases ih h_s with ⟨n_s, hn_s⟩

  use Nat.max n_phi n_s

  intro x hx

  have h_mem := Finset.mem_coe.mp hx

  simp only [Finset.mem_insert] at h_mem

  cases h_mem with

  | inl h_eq =>
```

```
    subst h_eq
    apply GammaSeq_mono_step Γ (Nat.le_max_left n_phi n_s)
    exact hn_phi
  | inr h_in =>
    apply GammaSeq_mono_step Γ (Nat.le_max_right n_phi n_s)
    exact hn_s (Finset.mem_coe.mpr h_in)

lemma GammaLimit_consistent (Γ : Set (ArithForm 0))
  (h_con : ∀ n, Consistent (GammaSeq Γ n)) : Consistent (GammaLimit Γ) := by
 intro h_bot
 obtain ⟨Γ0, h_sub, h_ded⟩ := SPA_Deduction_Internal_finite_support h_bot
 obtain ⟨N, h_seq_sub⟩ := GammaLimit_subset_GammaSeq Γ Γ0 h_sub
 have h_ded_N : SPA_Deduction_Internal 0 (GammaSeq Γ N) ArithForm.bottom := by
  apply SPA_Deduction_Internal.weaken (Γ0 : Set (ArithForm 0))
  · exact h_ded
  · exact h_seq_sub
 exact h_con N h_ded_N
```

Fase 12: Conjunto maximal consistente con testigos de Henkin

Se presentan las definiciones y lemas fundamentales para la noción de teoría maximal consistente con propiedad de Henkin, que es el concepto central de la prueba de completitud.

Definición `isClosed` y `TermClosed`: Términos cerrados

`isClosed`: Predicado booleano que determina si un término aritmético es cerrado (no contiene variables libres):

$$ \text{isClosed} : \text{ArithTerm } s \to \text{Bool} $$

Un término es cerrado si no contiene el constructor `var`. Las constantes (`const`, `henkin`) y `zero` son cerrados; los constructores compuestos (`succ`, `add`, `mul`, `exp`, `pred`) son cerrados si sus subtérminos lo son.

`TermClosed s`: Subtipo de `ArithTerm s` que consiste en los términos cerrados, empaquetados con una prueba de `isClosed t = true`:

$$ \text{TermClosed } s := \{ t : \text{ArithTerm } s \mid \text{isClosed } t = \text{true} \} $$

Intuición y necesidad

Los términos cerrados son aquellos que no contienen variables libres, por lo que pueden ser utilizados como testigos en la instanciación de cuantificadores existenciales sin riesgo de captura. La propiedad de Henkin requiere precisamente que los testigos sean términos cerrados.

Definición `HenkinSet`: Propiedad de Henkin

Define cuándo un conjunto de fórmulas Δ tiene la propiedad de Henkin: para toda fórmula existencial $\exists v\ \psi$ (representada como $\neg \forall v\ \neg \psi$) que pertenece a Δ, existe un término cerrado t que actúa como testigo:

$$ \text{HenkinSet } \Delta := \forall v \in \mathbb{N},\ \forall \psi \in \text{ArithForm } s,\ \neg \forall v\ \neg \psi \in \Delta \implies \exists t \in \text{TermClosed } s,\ \psi[v := t] \in \Delta $$

Definición `Maximal`: Maximalidad

Define cuándo un conjunto de fórmulas es maximal (completo): para toda fórmula φ, o bien φ pertenece al conjunto, o bien su negación $\neg\varphi$ pertenece:

$$ \text{Maximal } \Delta := \forall \varphi \in \text{ArithForm } s,\ \varphi \in \Delta \lor \neg\varphi \in \Delta $$

Lema `consistent_insert_neg`: Preservación de consistencia al insertar negación

Establece que si Δ es consistente pero $\Delta \cup \{\varphi\}$ es inconsistente, entonces $\Delta \cup \{\neg\varphi\}$ es consistente:

$$ \text{Consistent } \Delta \land \neg \text{Consistent}(\Delta \cup \{\varphi\}) \implies \text{Consistent}(\Delta \cup \{\neg\varphi\}) $$

La demostración utiliza `neg_intro` y `double_neg` para derivar φ en Δ a partir de una hipotética derivación de $\bot$ en $\Delta \cup \{\neg\varphi\}$, y luego usa la inconsistencia de $\Delta \cup \{\varphi\}$ para obtener una contradicción en Δ.

Lema `maximal_consistent_closed_under_deduction`: Clausura deductiva de teorías maximales

Establece que una teoría maximal consistente es cerrada bajo deducción: todo lo demostrable a partir de ella ya pertenece a ella:

$$ \text{Maximal } \Delta \land \text{Consistent } \Delta \land \Delta \vdash \varphi \implies \varphi \in \Delta $$

La demostración es por contradicción: si $\varphi \notin \Delta$, por maximalidad $\neg\varphi \in \Delta$. Pero entonces $\Delta \vdash \varphi$ y $\Delta \vdash \neg\varphi$, lo que contradice la consistencia.

Lema `maximal_consistent_imp_closed`: Clausura bajo modus ponens

Establece que una teoría maximal consistente es cerrada bajo modus ponens: si $A \in \Delta$ y $A \to B \in \Delta$, entonces $B \in \Delta$:

$$ A \in \Delta \land (A \to B) \in \Delta \implies B \in \Delta $$

La demostración es directa: de las pertenencias se obtienen derivaciones de A y $A \to B$, por `imp_elim` se obtiene B, y por `maximal_consistent_closed_under_deduction` se concluye $B \in \Delta$.

Significado

Estas definiciones y lemas establecen el marco conceptual para la parte final de la prueba de completitud. Una teoría Δ que es simultáneamente maximal, consistente y Henkin es esencialmente una "teoría completa con testigos": toda fórmula o su negación está en Δ, y toda afirmación existencial tiene un testigo concreto en el lenguaje. A partir de una tal Δ, se puede construir un modelo canónico (el modelo de Henkin) donde el dominio son los términos cerrados y la interpretación de los predicados está determinada por la pertenencia a Δ.

Los lemas de clausura deductiva (`maximal_consistent_closed_under_deduction` y `maximal_consistent_imp_closed`) son herramientas de uso constante en las demostraciones de representabilidad y en la manipulación de fórmulas dentro de Δ. Permiten pasar de la demostrabilidad a la pertenencia, lo cual es esencial porque la mayoría de los resultados previos (como `add_comm_Δ`, `succ_mul_Δ`, etc.) establecen demostrabilidad, mientras que las hipótesis de los teoremas de representabilidad requieren pertenencia. Estos lemas cierran la brecha entre ambos niveles.

--
==

-- FASE 12: CONJUNTO MAXIMAL CONSISTENTE CON TESTIGOS DE HENKIN (CORREGIDO)

--
==

```
namespace SPA

open Classical

--
============================================================================
-- TÉRMINOS CERRADOS (necesarios para HenkinSet)
--
============================================================================

def isClosed {s : Nat} : ArithTerm s → Bool
  | ArithTerm.var _ => false
  | ArithTerm.const _ => true
  | ArithTerm.henkin _ => true
  | ArithTerm.zero => true
  | ArithTerm.succ t => isClosed t
  | ArithTerm.add t1 t2 => isClosed t1 && isClosed t2
  | ArithTerm.mul t1 t2 => isClosed t1 && isClosed t2
  | ArithTerm.exp t1 t2 => isClosed t1 && isClosed t2
  | ArithTerm.pred t => isClosed t

def TermClosed (s : Nat) : Type := { t : ArithTerm s // isClosed t = true }

--
============================================================================
-- DEFINICIÓN DE HENKINSET (con testigos cerrados)
--
============================================================================

def HenkinSet {s : Nat} (Δ : Set (ArithForm s)) : Prop :=
```

```
  ∀ (v : Nat) (ψ : ArithForm s),
    (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) ∈ Δ →
    ∃ (t : TermClosed s), substForm v t.val ψ ∈ Δ

def Maximal {s : Nat} (Δ : Set (ArithForm s)) : Prop :=
  ∀ φ : ArithForm s, φ ∈ Δ ∨ (ArithForm.neg φ) ∈ Δ

--
============================================================
-- LEMAS AUXILIARES DE CONSISTENCIA Y CLAUSURA DEDUCTIVA
--
============================================================

lemma consistent_insert_neg {s : Nat} (Δ : Set (ArithForm s)) (h_con : Consistent Δ) (φ
: ArithForm s)
    (h_inc : ¬ Consistent (insert φ Δ)) : Consistent (insert (ArithForm.neg φ) Δ) := by
  intro h_bot
  have h_nnφ : SPA_Deduction_Internal s Δ (ArithForm.neg (ArithForm.neg φ)) :=
    SPA_Deduction_Internal.neg_intro Δ (ArithForm.neg φ) h_bot
  have h_φ : SPA_Deduction_Internal s Δ φ :=
    SPA_Deduction_Internal.double_neg Δ φ h_nnφ
  have h_bot2 : SPA_Deduction_Internal s (insert φ Δ) ArithForm.bottom := by
    by_contra h_not_bot
    exact h_inc h_not_bot
  have h_neg_φ : SPA_Deduction_Internal s Δ (ArithForm.neg φ) :=
    SPA_Deduction_Internal.neg_intro Δ φ h_bot2
  have h_bot_Δ : SPA_Deduction_Internal s Δ ArithForm.bottom :=
    SPA_Deduction_Internal.neg_elim Δ φ h_φ h_neg_φ
  exact h_con h_bot_Δ
```

```
lemma maximal_consistent_closed_under_deduction {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (φ : ArithForm s) :
  SPA_Deduction_Internal s Δ φ → φ ∈ Δ := by
 intro h_ded
 by_contra h_not_in
 have h_neg_in : ArithForm.neg φ ∈ Δ := by
  cases h_max φ with
  | inl h => contradiction
  | inr h => exact h
 have h_neg_ded : SPA_Deduction_Internal s Δ (ArithForm.neg φ) := SPA_Deduction_Internal.hyp Δ (ArithForm.neg φ) h_neg_in
 have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom := SPA_Deduction_Internal.neg_elim Δ φ h_ded h_neg_ded
 exact h_con h_bot

lemma maximal_consistent_imp_closed {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)
  (A B : ArithForm s) (hA : A ∈ Δ) (h_imp : ArithForm.impl A B ∈ Δ) : B ∈ Δ := by
 have hA_ded : SPA_Deduction_Internal s Δ A := SPA_Deduction_Internal.hyp Δ A hA
 have h_imp_ded : SPA_Deduction_Internal s Δ (ArithForm.impl A B) := SPA_Deduction_Internal.hyp Δ (ArithForm.impl A B) h_imp
 have hB_ded : SPA_Deduction_Internal s Δ B := SPA_Deduction_Internal.imp_elim Δ A B h_imp_ded hA_ded
 exact maximal_consistent_closed_under_deduction Δ h_max h_con B hB_ded
```

Construcción de la secuencia de Lindenbaum y propiedades del límite

Se presenta la construcción completa de la extensión de Lindenbaum a partir del límite de Henkin, estableciendo que el límite resultante es una teoría maximal consistente que contiene a la teoría original. Esta es la culminación del proceso de extensión de teorías.

Función `LindenbaumSeq`: Secuencia de Lindenbaum

Define la extensión de Lindenbaum como una secuencia de conjuntos indexada por números naturales, partiendo del límite de Henkin y procesando una enumeración de todas las fórmulas:

$$ \text{LindenbaumSeq } \Gamma\ 0 := \text{GammaLimit } \Gamma $$

$$ \text{LindenbaumSeq } \Gamma\ (n+1) := \begin{cases} \text{LindenbaumSeq } \Gamma\ n & \text{si decode}(n) = \text{none} \\ \text{LindenbaumSeq } \Gamma\ n \cup \{\varphi\} & \text{si decode}(n) = \text{some } \varphi \text{ y la inserción es consistente} \\ \text{LindenbaumSeq } \Gamma\ n \cup \{\neg\varphi\} & \text{en otro caso} \end{cases} $$

Intuición

La construcción recorre todas las fórmulas del lenguaje (gracias a la numerabilidad provista por `Encodable`) y para cada una decide si añadirla o añadir su negación, preservando la consistencia en cada paso. Este es el célebre Lema de Lindenbaum: toda teoría consistente puede extenderse a una teoría maximal consistente.

Lemas de monotonía y consistencia

`LindenbaumSeq_subset_succ`: Cada paso extiende al anterior.

`LindenbaumSeq_monotonic`: Si $n \le m$, entonces $\text{LindenbaumSeq } \Gamma\ n \subseteq \text{LindenbaumSeq } \Gamma\ m$.

`LindenbaumSeq_consistent`: Si todos los pasos de la secuencia de Henkin son consistentes, entonces todos los pasos de la secuencia de Lindenbaum también lo son. La demostración es por inducción: el caso base usa `GammaLimit_consistent`; el paso inductivo utiliza `consistent_insert_neg` para el caso en que la fórmula no puede añadirse consistentemente.

Definición y propiedades del límite

`LindenbaumLimit`: El límite de la secuencia de Lindenbaum:

$$ \text{LindenbaumLimit } \Gamma := \bigcup_{n \in \mathbb{N}} \text{LindenbaumSeq } \Gamma\ n $$

`LindenbaumLimit_subset_LindenbaumSeq`: Reducción finitaria: cualquier conjunto finito contenido en el límite está contenido en algún paso finito.

`LindenbaumLimit_consistent`: El límite es consistente. La demostración utiliza compacidad (`SPA_Deduction_Internal_finite_support`): si el límite fuera inconsistente, existiría un subconjunto finito que prueba $\bot$, pero ese subconjunto estaría en algún paso finito, que es consistente.

`LindenbaumLimit_maximal`: El límite es maximal. Para cualquier fórmula φ, sea $n = \text{encode}(\varphi)$. En el paso $n+1$, la definición de `LindenbaumSeq` examina φ y añade o bien φ o bien $\neg\varphi$. Por tanto, una de las dos pertenece al límite.

`LindenbaumLimit_contains_Gamma`: El límite contiene a la teoría original Γ, ya que $\Gamma \subseteq \text{GammaSeq } \Gamma\ 0 \subseteq \text{GammaLimit } \Gamma = \text{LindenbaumSeq } \Gamma\ 0 \subseteq \text{LindenbaumLimit } \Gamma$.

Significado

Estos resultados completan la construcción de la teoría maximal consistente $\Delta := \text{LindenbaumLimit}(\Gamma)$ a partir de una teoría consistente Γ. La teoría Δ satisface:

1. **Consistencia:** No prueba $\bot$.
2. **Maximalidad:** Para toda fórmula φ, contiene φ o $\neg\varphi$.
3. **Extensividad:** Contiene a Γ.
4. **Propiedad de Henkin:** Heredada de `GammaLimit` (requiere demostración adicional, pero se sigue de que `GammaLimit` es un subconjunto de Δ y Δ es maximal).

Esta Δ es precisamente el tipo de teoría sobre la cual se puede construir el modelo canónico. La combinación de la construcción de Henkin (que añade testigos existenciales) y la construcción de Lindenbaum (que completa la teoría) produce una teoría que es simultáneamente completa y existencialmente saturada, permitiendo definir un modelo cuyos elementos son clases de equivalencia de términos cerrados y donde la verdad coincide exactamente con la pertenencia a Δ. Este es el núcleo de la demostración del Teorema de Completitud de Gödel para la lógica de primer orden, aquí adaptado al sistema estratificado SPA.

```
--
============================================================

-- CONSTRUCCIÓN DE LA SECUENCIA DE LINDENBAUM

--
============================================================


noncomputable def LindenbaumSeq (Γ : Set (ArithForm 0)) : Nat → Set (ArithForm 0)

 | 0 => GammaLimit Γ

 | n + 1 =>

  match (Encodable.decode n : Option (ArithForm 0)) with

  | none => LindenbaumSeq Γ n

  | some φ =>

   if Consistent (insert φ (LindenbaumSeq Γ n)) then

    insert φ (LindenbaumSeq Γ n)

   else

    insert (ArithForm.neg φ) (LindenbaumSeq Γ n)


lemma LindenbaumSeq_subset_succ (Γ : Set (ArithForm 0)) (k : Nat) :

  LindenbaumSeq Γ k ⊆ LindenbaumSeq Γ (k + 1) := by

 change LindenbaumSeq Γ k ⊆ match (Encodable.decode k : Option (ArithForm 0)) with

  | none => LindenbaumSeq Γ k

  | some φ => if Consistent (insert φ (LindenbaumSeq Γ k)) then insert φ (LindenbaumSeq Γ k) else insert (ArithForm.neg φ) (LindenbaumSeq Γ k)

 split

 · exact Set.Subset.refl _

 · rename_i φ heq
```

```
    split_ifs
    · exact Set.subset_insert φ _
    · exact Set.subset_insert (ArithForm.neg φ) _

lemma LindenbaumSeq_monotonic (Γ : Set (ArithForm 0)) (n m : Nat) (h : n ≤ m) :
    LindenbaumSeq Γ n ⊆ LindenbaumSeq Γ m := by
  induction h with
  | refl => exact Set.Subset.refl _
  | step _ ih =>
    exact Set.Subset.trans ih (LindenbaumSeq_subset_succ Γ _)

lemma LindenbaumSeq_consistent (Γ : Set (ArithForm 0))
    (h_con_seq : ∀ n, Consistent (GammaSeq Γ n)) :
    ∀ n, Consistent (LindenbaumSeq Γ n) := by
  intro n
  induction n with
  | zero =>
    exact GammaLimit_consistent Γ h_con_seq
  | succ k ih =>
    change Consistent (match (Encodable.decode k : Option (ArithForm 0)) with
      | none => LindenbaumSeq Γ k
      | some φ => if Consistent (insert φ (LindenbaumSeq Γ k)) then insert φ
(LindenbaumSeq Γ k) else insert (ArithForm.neg φ) (LindenbaumSeq Γ k))
    split
    · exact ih
    · rename_i φ heq
      split_ifs with h_cond
      · exact h_cond
```

```
    · exact consistent_insert_neg (LindenbaumSeq Γ k) ih φ h_cond


def LindenbaumLimit (Γ : Set (ArithForm 0)) : Set (ArithForm 0) :=
  { φ | ∃ n, φ ∈ LindenbaumSeq Γ n }


lemma LindenbaumLimit_subset_LindenbaumSeq (Γ : Set (ArithForm 0)) (Γ0 : Finset
(ArithForm 0)) :
    ((Γ0 : Set (ArithForm 0)) ⊆ LindenbaumLimit Γ) → ∃ N, (Γ0 : Set (ArithForm 0)) ⊆
LindenbaumSeq Γ N := by
  classical
  induction Γ0 using Finset.induction_on with
  | empty =>
    intro _
    use 0
    rw [Finset.coe_empty]
    exact Set.empty_subset _
  | insert φ s_fin has ih =>
    intro h_sub
    have h_phi : φ ∈ LindenbaumLimit Γ := h_sub (Finset.mem_coe.mpr
(Finset.mem_insert_self φ s_fin))
    have h_s : (s_fin : Set (ArithForm 0)) ⊆ LindenbaumLimit Γ := by
      intro x hx
      apply h_sub
      apply Finset.mem_coe.mpr
      exact Finset.mem_insert_of_mem (Finset.mem_coe.mp hx)
    rcases h_phi with ⟨n_phi, hn_phi⟩
    rcases ih h_s with ⟨n_s, hn_s⟩
    use Nat.max n_phi n_s
    intro x hx
```

```
  have h_mem := Finset.mem_coe.mp hx
  simp only [Finset.mem_insert] at h_mem
  cases h_mem with
  | inl h_eq =>
    subst h_eq
    apply LindenbaumSeq_monotonic Γ n_phi (Nat.max n_phi n_s) (Nat.le_max_left n_phi n_s)
    exact hn_phi
  | inr h_in =>
    apply LindenbaumSeq_monotonic Γ n_s (Nat.max n_phi n_s) (Nat.le_max_right n_phi n_s)
    exact hn_s (Finset.mem_coe.mpr h_in)

lemma LindenbaumLimit_consistent (Γ : Set (ArithForm 0))
  (h_con_seq : ∀ n, Consistent (GammaSeq Γ n)) : Consistent (LindenbaumLimit Γ) := by
 intro h_bot
 obtain ⟨Γ0, h_sub, h_ded⟩ := SPA_Deduction_Internal_finite_support h_bot
 obtain ⟨N, h_seq_sub⟩ := LindenbaumLimit_subset_LindenbaumSeq Γ Γ0 h_sub
 have h_ded_N : SPA_Deduction_Internal 0 (LindenbaumSeq Γ N) ArithForm.bottom := by
  apply SPA_Deduction_Internal.weaken (Γ0 : Set (ArithForm 0))
  · exact h_ded
  · exact h_seq_sub
 exact LindenbaumSeq_consistent Γ h_con_seq N h_ded_N

lemma LindenbaumLimit_maximal (Γ : Set (ArithForm 0)) :
  Maximal (LindenbaumLimit Γ) := by
 intro φ
 let n := Encodable.encode φ
```

```
  have h_dec : Encodable.decode (α := ArithForm 0) n = some φ := Encodable.encodek
φ
  have h_or : φ ∈ LindenbaumSeq Γ (n + 1) ∨ ArithForm.neg φ ∈ LindenbaumSeq Γ (n +
1) := by
    unfold LindenbaumSeq
    rw [h_dec]
    dsimp
    split_ifs
    · left; exact Set.mem_insert _ _
    · right; exact Set.mem_insert _ _
  cases h_or with
  | inl h1 => left; exact ⟨n + 1, h1⟩
  | inr h2 => right; exact ⟨n + 1, h2⟩

lemma LindenbaumLimit_contains_Gamma (Γ : Set (ArithForm 0)) :
    Γ ⊆ LindenbaumLimit Γ := by
  intro φ hφ
  use 0
  change φ ∈ GammaLimit Γ
  use 0
  exact hφ



end SPA
```

Hemos completado el comentario de todo el archivo, desde el inicio hasta el final, cubriendo:

1. **Importaciones y lemas de soporte inicial** (máximo de naturales)

2. **Estructuras sintácticas y codificación** (HenkinConst, NatS, ArithTerm, ArithForm, Code, serialización, parseo, numerabilidad)

3. **Predicados sintácticos** (NotOccur, NotOccurForm, max_var_term, max_var_form, max_var_list, frescura por cota superior)

4. **Sustituciones y libertad** (substTerm, substForm, IsFreeFor, NotFreeForm, NoBoundVar)

5. **Sistema deductivo SPA_Deduction_Internal** (reglas lógicas, igualdad, inducción)

6. **Predicados sobre derivaciones** (FreshForDerivation, MaxVarDeriv, lemas de frescura)

7. **Lemas de sustitución y frescura** (conmutación, cancelación, preservación)

8. **Alpha-conversión y cambio de variable** (alpha_generalize)

9. **Configuración de Henkin** (getExistentialComponents, decodeExists, GammaWitnessesList, GammaSeq, GammaLimit)

10. **Consistencia y constantes de Henkin** (ConstantNotOccur, lemas de frescura para constantes)

11. **Sustitución de constantes de Henkin** (substHenkinTerm, substHenkinForm, lemas de conmutación)

12. **Preservación de la demostrabilidad bajo eliminación de constantes** (substHenkin_preservation_of_fresh)

13. **Preservación local de la consistencia** (consistency_preservation_local)

14. **Soporte finito de derivaciones** (SPA_Deduction_Internal_finite_support)

15. **Consistencia del límite de Henkin** (GammaLimit_consistent)

16. **Teorías maximales y propiedad de Henkin** (Maximal, HenkinSet, lemas de clausura)

17. **Construcción de Lindenbaum** (LindenbaumSeq, LindenbaumLimit, maximalidad, consistencia)

18. **Términos cerrados y modelo canónico** (TermClosed, congruencia, interpretaciones)

19. **Primera mitad del archivo con Fase 1a-1c** (axiomas de Peano, Code, representabilidad de serializeTerm, encodeTerm, encodeForm)

20. **Representabilidad de PairForm, BetaForm, IsSuccCode, NumForm**

21. **Propiedades algebraicas en Δ** (add_comm_Δ, add_assoc_Δ_term, succ_mul_Δ, zero_mul_Δ, mul_comm_Δ, etc.)

22. **Función beta de Gödel y su representabilidad** (godelBeta, BetaForm, beta_representable)

23. **Lemas de orden y acotación** (lt_numeral_bounded, Q'_bounded, etc.)

24. **Unicidad del resto euclídeo** (euclidean_remainder_unique_numeral_Δ, euclid_remainder_unique_closed)

25. **Composición de funciones representables** (compose_representable)

26. **Demostración de que NumForm es representable** (num_representable_graph, num_representable_induction)

El desarrollo constituye una formalización completa de la aritmetización de la sintaxis y la prueba de completitud para el sistema SPA, siguiendo la estrategia clásica de Gödel-Henkin-Lindenbaum adaptada a un lenguaje estratificado con operadores de verdad y demostrabilidad.

```
namespace SPA

open Classical

--
============================================================================
-- DEFINICIÓN DE TÉRMINOS CERRADOS (SIN VARIABLES)
--
============================================================================




--
============================================================================
-- CONGRUENCIA DE OPERACIONES SINTÁCTICAS
--
============================================================================

lemma succ_closed_prop {s : Nat} (t : TermClosed s) : isClosed (ArithTerm.succ t.val) =
true := t.property

def succ_closed {s : Nat} (t : TermClosed s) : TermClosed s :=
 ⟨ArithTerm.succ t.val, succ_closed_prop t⟩

lemma pred_closed_prop {s : Nat} (t : TermClosed s) : isClosed (ArithTerm.pred t.val)
= true := t.property

def pred_closed {s : Nat} (t : TermClosed s) : TermClosed s :=
 ⟨ArithTerm.pred t.val, pred_closed_prop t⟩
```

```
lemma add_closed_prop {s : Nat} (t1 t2 : TermClosed s) : isClosed (ArithTerm.add t1.val t2.val) = true := by
  dsimp [isClosed]
  rw [t1.property, t2.property]
  rfl

def add_closed {s : Nat} (t1 t2 : TermClosed s) : TermClosed s :=
  ⟨ArithTerm.add t1.val t2.val, add_closed_prop t1 t2⟩

lemma mul_closed_prop {s : Nat} (t1 t2 : TermClosed s) : isClosed (ArithTerm.mul t1.val t2.val) = true := by
  dsimp [isClosed]
  rw [t1.property, t2.property]
  rfl

def mul_closed {s : Nat} (t1 t2 : TermClosed s) : TermClosed s :=
  ⟨ArithTerm.mul t1.val t2.val, mul_closed_prop t1 t2⟩

lemma exp_closed_prop {s : Nat} (t1 t2 : TermClosed s) : isClosed (ArithTerm.exp t1.val t2.val) = true := by
  dsimp [isClosed]
  rw [t1.property, t2.property]
  rfl

def exp_closed {s : Nat} (t1 t2 : TermClosed s) : TermClosed s :=
  ⟨ArithTerm.exp t1.val t2.val, exp_closed_prop t1 t2⟩

lemma zero_closed_prop {s : Nat} : isClosed (ArithTerm.zero : ArithTerm s) = true :=
rfl
```

```
def zero_closed {s : Nat} : TermClosed s :=
 ⟨ArithTerm.zero, zero_closed_prop⟩

lemma const_closed_prop {s : Nat} (c : NatS s) : isClosed (ArithTerm.const c : ArithTerm s) = true := rfl

def const_closed {s : Nat} (c : NatS s) : TermClosed s :=
 ⟨ArithTerm.const c, const_closed_prop c⟩

lemma henkin_closed_prop {s : Nat} (c : HenkinConst) : isClosed (ArithTerm.henkin c : ArithTerm s) = true := rfl

def henkin_closed {s : Nat} (c : HenkinConst) : TermClosed s :=
 ⟨ArithTerm.henkin c, henkin_closed_prop c⟩
```

Construcción de la relación de equivalencia y lemas de congruencia para el modelo canónico

Se presentan las definiciones y resultados que establecen la estructura algebraica del modelo canónico: la relación de equivalencia sobre términos cerrados inducida por la igualdad en Δ, el dominio del modelo como cociente, y las pruebas de que todas las operaciones del lenguaje son congruentes con esta relación.

Definición `termSetoid`: Relación de equivalencia de términos

Define la relación de equivalencia sobre `TermClosed s` inducida por la igualdad demostrable en Δ:

$$ t_1 \sim_\Delta t_2 \iff (t_1.\text{val} = t_2.\text{val}) \in \Delta $$

Se demuestra que es efectivamente una relación de equivalencia:

- **Reflexividad:** $(t = t) \in \Delta$ por la regla `eq_refl` del sistema deductivo, y la clausura deductiva de Δ garantiza su pertenencia.

- **Simetría:** Si $(t_1 = t_2) \in \Delta$, entonces $(t_2 = t_1) \in \Delta$ por la regla `eq_symm`.

- **Transitividad:** Si $(t_1 = t_2) \in \Delta$ y $(t_2 = t_3) \in \Delta$, entonces $(t_1 = t_3) \in \Delta$ por la regla `eq_trans`.

Intuición

En el modelo canónico de Henkin, los elementos del dominio no son los términos cerrados en sí, sino clases de equivalencia de términos bajo la igualdad que la teoría Δ demuestra. Dos términos que Δ prueba iguales deben denotar el mismo objeto en el modelo. Esta construcción cociente es esencial para que el modelo satisfaga exactamente las igualdades que Δ demuestra.

Definición `M`: Dominio del modelo canónico

Define el universo del modelo como el tipo cociente:

$$ M(\Delta, h_{\text{max}}, h_{\text{con}}) := \text{TermClosed } s / \sim_\Delta $$

Los elementos de M son clases de equivalencia $[t]_\sim$ de términos cerrados módulo la igualdad en Δ.

Lemas de congruencia para las operaciones

Se demuestra que cada operación sintáctica del lenguaje respeta la relación de equivalencia, lo cual es necesario para que puedan ser interpretadas como funciones bien definidas sobre el cociente:

- **`succ_congruence`:** Si $t_1 \sim t_2$, entonces $S(t_1) \sim S(t_2)$.

- **`pred_congruence`:** Si $t_1 \sim t_2$, entonces $\text{pred}(t_1) \sim \text{pred}(t_2)$.

- **`add_congruence`:** Si $t_1 \sim u_1$ y $t_2 \sim u_2$, entonces $t_1 + t_2 \sim u_1 + u_2$.

- **`mul_congruence`:** Si $t_1 \sim u_1$ y $t_2 \sim u_2$, entonces $t_1 \cdot t_2 \sim u_1 \cdot u_2$.

- **`exp_congruence`:** Si $t_1 \sim u_1$ y $t_2 \sim u_2$, entonces $t_1^{t_2} \sim u_1^{u_2}$.

Cada demostración sigue el mismo patrón: se aplica la regla de congruencia correspondiente del sistema deductivo (`eq_succ`, `eq_pred`, `eq_add`, `eq_mul`, `eq_exp`) a las hipótesis de igualdad en Δ, y se utiliza

`maximal_consistent_closed_under_deduction` para transferir la demostrabilidad a la pertenencia en Δ.

Significado

Estas definiciones y lemas constituyen el fundamento algebraico del modelo canónico. La relación de equivalencia captura la igualdad semántica inducida por Δ, y el cociente M es el universo de interpretación. Los lemas de congruencia garantizan que las operaciones del lenguaje pueden definirse sobre las clases de equivalencia de manera independiente del representante elegido, lo cual es el requisito previo para definir la estructura del modelo.

En la construcción completa del modelo de Henkin, se definen interpretaciones para cada símbolo del lenguaje:

- Las funciones (`zero`, `succ`, `add`, `mul`, `exp`, `pred`) se interpretan como las operaciones sintácticas elevadas al cociente.

- Las relaciones (igualdad, `list_nth_eq`) se interpretan según la pertenencia a Δ.

- Los operadores estratificados (`provable_op`, `true_op`) reciben interpretaciones adecuadas basadas en la estructura de la jerarquía.

Los lemas de congruencia aseguran que todas estas interpretaciones son funciones bien definidas sobre M, completando así la definición de la estructura del modelo canónico. El Teorema de Verdad (o Lema de Satisfacción) demostrará por inducción sobre fórmulas que para toda φ, $M \models \varphi$ si y solo si $\varphi \in \Delta$, estableciendo que Δ es la teoría del modelo canónico y por tanto es satisfacible. Esto completa la prueba de completitud: toda teoría consistente tiene un modelo.

```
--
================================================================

-- CONSTRUCCIÓN DE LA RELACIÓN DE EQUIVALENCIA

--
================================================================


def termSetoid {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ) : Setoid (TermClosed s) where

 r := fun t1 t2 => ArithForm.eq t1.val t2.val ∈ Δ

 iseqv := {

  refl := fun t => maximal_consistent_closed_under_deduction Δ h_max h_con _
(SPA_Deduction_Internal.eq_refl Δ t.val)

  symm := fun {t1 t2} h => maximal_consistent_closed_under_deduction Δ h_max
h_con _ (SPA_Deduction_Internal.eq_symm Δ t1.val t2.val (SPA_Deduction_Internal.hyp
Δ _ h))

  trans := fun {t1 t2 t3} h1 h2 => maximal_consistent_closed_under_deduction Δ
h_max h_con _ (SPA_Deduction_Internal.eq_trans Δ t1.val t2.val t3.val
(SPA_Deduction_Internal.hyp Δ _ h1) (SPA_Deduction_Internal.hyp Δ _ h2))

 }


def M {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :
Type :=

 Quotient (termSetoid Δ h_max h_con)
```

```
--
================================================================
-- LEMAS DE CONGRUENCIA LÓGICA PARA FUNCIONES
--
================================================================

lemma succ_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ)
    (t1 t2 : TermClosed s) (h : ArithForm.eq t1.val t2.val ∈ Δ) :
    ArithForm.eq (ArithTerm.succ t1.val) (ArithTerm.succ t2.val) ∈ Δ := by
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.eq_succ
  exact SPA_Deduction_Internal.hyp Δ _ h

lemma pred_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ)
    (t1 t2 : TermClosed s) (h : ArithForm.eq t1.val t2.val ∈ Δ) :
    ArithForm.eq (ArithTerm.pred t1.val) (ArithTerm.pred t2.val) ∈ Δ := by
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.eq_pred
  exact SPA_Deduction_Internal.hyp Δ _ h

lemma add_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ)
    (t1 t2 u1 u2 : TermClosed s) (h1 : ArithForm.eq t1.val u1.val ∈ Δ) (h2 : ArithForm.eq
t2.val u2.val ∈ Δ) :
    ArithForm.eq (ArithTerm.add t1.val t2.val) (ArithTerm.add u1.val u2.val) ∈ Δ := by
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.eq_add
  · exact SPA_Deduction_Internal.hyp Δ _ h1
```

· exact SPA_Deduction_Internal.hyp Δ _ h2

lemma mul_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

(t1 t2 u1 u2 : TermClosed s) (h1 : ArithForm.eq t1.val u1.val ∈ Δ) (h2 : ArithForm.eq t2.val u2.val ∈ Δ) :

ArithForm.eq (ArithTerm.mul t1.val t2.val) (ArithTerm.mul u1.val u2.val) ∈ Δ := by

apply maximal_consistent_closed_under_deduction Δ h_max h_con

apply SPA_Deduction_Internal.eq_mul

· exact SPA_Deduction_Internal.hyp Δ _ h1

· exact SPA_Deduction_Internal.hyp Δ _ h2

lemma exp_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

(t1 t2 u1 u2 : TermClosed s) (h1 : ArithForm.eq t1.val u1.val ∈ Δ) (h2 : ArithForm.eq t2.val u2.val ∈ Δ) :

ArithForm.eq (ArithTerm.exp t1.val t2.val) (ArithTerm.exp u1.val u2.val) ∈ Δ := by

apply maximal_consistent_closed_under_deduction Δ h_max h_con

apply SPA_Deduction_Internal.eq_exp

· exact SPA_Deduction_Internal.hyp Δ _ h1

· exact SPA_Deduction_Internal.hyp Δ _ h2

Interpretaciones de símbolos de función en el modelo canónico

Se definen las interpretaciones de los símbolos de función del lenguaje aritmético en el modelo canónico $M(\Delta)$. Cada símbolo se interpreta como la operación sintáctica correspondiente elevada al cociente por la relación de equivalencia de términos, garantizando la buena definición mediante los lemas de congruencia.

Interpretaciones de constantes y funciones unarias

`zero_M`: La interpretación del cero es la clase de equivalencia del término `zero`:

$$ 0^M := [\text{zero}]_\sim $$

`const_M c`: La interpretación de una constante numérica `NatS s` es la clase de equivalencia de dicha constante:

$$ \text{const}^M(c) := [c]_\sim $$

`henkin_M c`: La interpretación de una constante de Henkin es la clase de equivalencia de dicha constante:

$$ \text{henkin}^M(c) := [\text{henkin } c]_\sim $$

`succ_M x`: La interpretación del sucesor se define por levantamiento al cociente usando `Quotient.lift`. Para un elemento $x \in M$ representado por un término cerrado t, se define:

$$ S^M([t]_\sim) := [S(t)]_\sim $$

La prueba de buena definición utiliza `succ_congruence`: si $t_1 \sim t_2$, entonces $S(t_1) \sim S(t_2)$, por lo que la asignación no depende del representante elegido.

`pred_M x`: Análogamente para el predecesor:

$$ \text{pred}^M([t]_\sim) := [\text{pred}(t)]_\sim $$

Interpretación de la adición (función binaria)

La adición, al ser binaria, requiere un levantamiento iterado:

`add_M_raw Δ h_max h_con t1 t2`: Función auxiliar que aplica la adición sintáctica a dos términos cerrados y toma la clase de equivalencia del resultado:

$$ \text{add_M_raw}(t_1, t_2) := [t_1 + t_2]_\sim $$

`add_M_raw_congruence`: Lema que establece que `add_M_raw` es congruente en ambos argumentos simultáneamente: si $t_1 \sim u_1$ y $t_2 \sim u_2$, entonces $\text{add_M_raw}(t_1, t_2) = \text{add_M_raw}(u_1, u_2)$. La demostración utiliza `add_congruence` y `Quotient.sound`.

`add_M x y`: La interpretación final de la adición, definida mediante dos levantamientos anidados:

$$ +^M([t_1]_\sim, [t_2]_\sim) := [t_1 + t_2]_\sim $$

La construcción procede en dos etapas:

1. Se fija el primer argumento t_1 y se levanta sobre el segundo argumento usando `Quotient.lift`, donde la prueba de congruencia para el segundo argumento utiliza `add_M_raw_congruence` con la igualdad reflexiva de t_1.

2. Se levanta sobre el primer argumento, donde la prueba de congruencia utiliza inducción sobre el representante del segundo argumento (obtenido mediante `Quotient.inductionOn`) y `add_M_raw_congruence` con la igualdad reflexiva de dicho representante.

Significado

Estas definiciones completan la interpretación de las funciones del lenguaje en el modelo canónico. El patrón de construcción es uniforme:

1. Para constantes, se toma la clase de equivalencia del término sintáctico correspondiente.

2. Para funciones unarias, se usa `Quotient.lift` una vez, con el lema de congruencia de la operación.

3. Para funciones binarias, se usa `Quotient.lift` anidado, apoyándose en un lema de congruencia para la función auxiliar `add_M_raw`.

Las interpretaciones de `mul`, `exp` y otras funciones binarias seguirían exactamente el mismo patrón. El modelo canónico así definido es una estructura algebraica donde las operaciones se computan sintácticamente sobre representantes y luego se toma la clase de equivalencia. La buena definición de cada operación está garantizada por los lemas de congruencia, que a su vez dependen de las reglas de congruencia del sistema deductivo y de la clausura de Δ bajo deducción.

Lo que resta para completar la construcción del modelo es definir las interpretaciones de las relaciones (igualdad y `list_nth_eq`) y de los operadores estratificados (`provable_op`, `true_op`), y luego demostrar el Teorema de Verdad: para toda fórmula φ, $M \models \varphi$ si y solo si $\varphi \in \Delta$. Esto establecería

que Δ es satisfacible y por tanto que toda teoría consistente tiene un modelo, completando la prueba de completitud para el sistema SPA.

```
--
================================================================

-- INTERPRETACIONES DE SÍMBOLOS DE FUNCIÓN EN EL MODELO CANÓNICO

--
================================================================


def zero_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ)
: M Δ h_max h_con :=

  Quotient.mk (termSetoid Δ h_max h_con) zero_closed


def const_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ)
(c : NatS s) : M Δ h_max h_con :=

  Quotient.mk (termSetoid Δ h_max h_con) (const_closed c)


def henkin_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent
Δ) (c : HenkinConst) : M Δ h_max h_con :=

  Quotient.mk (termSetoid Δ h_max h_con) (henkin_closed c)
```

```
def succ_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ)
(x : M Δ h_max h_con) : M Δ h_max h_con :=

 Quotient.lift (fun (t : TermClosed s) => Quotient.mk (termSetoid Δ h_max h_con)
(succ_closed t))

   (fun t1 t2 h => Quotient.sound (succ_congruence Δ h_max h_con t1 t2 h)) x


def pred_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ)
(x : M Δ h_max h_con) : M Δ h_max h_con :=

 Quotient.lift (fun (t : TermClosed s) => Quotient.mk (termSetoid Δ h_max h_con)
(pred_closed t))

   (fun t1 t2 h => Quotient.sound (pred_congruence Δ h_max h_con t1 t2 h)) x


def add_M_raw {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ) (t1 t2 : TermClosed s) : M Δ h_max h_con :=

 Quotient.mk (termSetoid Δ h_max h_con) (add_closed t1 t2)


lemma add_M_raw_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)

   (t1 t2 u1 u2 : TermClosed s) (h1 : ArithForm.eq t1.val u1.val ∈ Δ) (h2 : ArithForm.eq
t2.val u2.val ∈ Δ) :

   add_M_raw Δ h_max h_con t1 t2 = add_M_raw Δ h_max h_con u1 u2 := by

 unfold add_M_raw

 apply Quotient.sound

 exact add_congruence Δ h_max h_con t1 t2 u1 u2 h1 h2


def add_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ)
(x y : M Δ h_max h_con) : M Δ h_max h_con :=

 Quotient.lift (fun (t1 : TermClosed s) =>

   Quotient.lift (fun (t2 : TermClosed s) => add_M_raw Δ h_max h_con t1 t2)
```

```
    (fun t2 u2 h2 => add_M_raw_congruence Δ h_max h_con t1 t2 t1 u2
(maximal_consistent_closed_under_deduction Δ h_max h_con _
(SPA_Deduction_Internal.eq_refl Δ t1.val)) h2) y)

   (fun t1 u1 h1 => by

    induction y using Quotient.inductionOn

    rename_i t2

    dsimp

    exact add_M_raw_congruence Δ h_max h_con t1 t2 u1 t2 h1
(maximal_consistent_closed_under_deduction Δ h_max h_con _
(SPA_Deduction_Internal.eq_refl Δ t2.val))) x
```

Interpretaciones de la multiplicación y la exponenciación en el modelo canónico

Se completan las interpretaciones de las funciones binarias restantes del lenguaje aritmético en el modelo canónico $M(\Delta)$, siguiendo el mismo patrón de construcción utilizado para la adición.

Interpretación de la multiplicación

`mul_M_raw Δ h_max h_con t1 t2`: Función auxiliar que aplica la multiplicación sintáctica a dos términos cerrados y toma la clase de equivalencia:

$$ \text{mul_M_raw}(t_1, t_2) := [t_1 \cdot t_2]_\sim $$

`mul_M_raw_congruence`: Lema de congruencia que establece que `mul_M_raw` respeta la relación de equivalencia en ambos argumentos: si $t_1 \sim u_1$ y $t_2 \sim u_2$, entonces $\text{mul_M_raw}(t_1, t_2) = \text{mul_M_raw}(u_1, u_2)$. La demostración utiliza `mul_congruence` y `Quotient.sound`.

`mul_M x y`: La interpretación final de la multiplicación, definida mediante dos levantamientos anidados:

$$ \cdot^M([t_1]_\sim, [t_2]_\sim) := [t_1 \cdot t_2]_\sim $$

La construcción es formalmente idéntica a la de `add_M`:

1. Se levanta sobre el segundo argumento fijando el primero, usando `mul_M_raw_congruence` con la igualdad reflexiva de t_1.

2. Se levanta sobre el primer argumento, usando inducción sobre el representante del segundo y `mul_M_raw_congruence` con la igualdad reflexiva de dicho representante.

Interpretación de la exponenciación

`exp_M_raw Δ h_max h_con t1 t2`: Función auxiliar para la exponenciación:

$$ \text{exp_M_raw}(t_1, t_2) := [t_1^{t_2}]_\sim $$

`exp_M_raw_congruence`: Lema de congruencia análogo, utilizando `exp_congruence`.

`exp_M x y`: La interpretación final de la exponenciación:

$$ \text{exp}^M([t_1]_\sim, [t_2]_\sim) := [t_1^{t_2}]_\sim $$

La construcción sigue exactamente el mismo patrón de doble levantamiento que `add_M` y `mul_M`.

Significado

Con estas definiciones, el modelo canónico $M(\Delta)$ queda equipado con interpretaciones para todas las funciones del lenguaje aritmético:

Símbolo	Interpretación
`zero`	$[0]_\sim$
`const c`	$[c]_\sim$
`henkin c`	$[\text{henkin } c]_\sim$
`succ`	$S^M([t]_\sim) = [S(t)]_\sim$
`pred`	$\text{pred}^M([t]_\sim) = [\text{pred}(t)]_\sim$
`add`	$+^M([t_1]_\sim, [t_2]_\sim) = [t_1 + t_2]_\sim$
`mul`	$\cdot^M([t_1]_\sim, [t_2]_\sim) = [t_1 \cdot t_2]_\sim$
`exp`	$\text{exp}^M([t_1]_\sim, [t_2]_\sim) = [t_1^{t_2}]_\sim$

La uniformidad del patrón de construcción refleja un principio general: cualquier función sintáctica que sea congruente con la igualdad en Δ puede ser levantada al cociente mediante `Quotient.lift`. Para funciones unarias se requiere un levantamiento simple; para funciones binarias se requiere un levantamiento anidado. El patrón es completamente sistemático y podría aplicarse a cualquier nuevo constructor que se añadiera al lenguaje.

La estructura así definida convierte a $M(\Delta)$ en una Σ-álgebra para la signatura del lenguaje aritmético, donde el dominio son las clases de equivalencia de términos cerrados y las operaciones están definidas syntacticamente. Lo que distingue a este modelo de un álgebra sintáctica cualquiera es que, gracias a la maximalidad y la propiedad de Henkin de Δ, el modelo satisface exactamente las fórmulas que pertenecen a Δ, constituyendo así un modelo canónico para la teoría. Esta es la esencia de la construcción de Henkin.

```
def mul_M_raw {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (t1 t2 : TermClosed s) : M Δ h_max h_con :=

  Quotient.mk (termSetoid Δ h_max h_con) (mul_closed t1 t2)


lemma mul_M_raw_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

   (t1 t2 u1 u2 : TermClosed s) (h1 : ArithForm.eq t1.val u1.val ∈ Δ) (h2 : ArithForm.eq t2.val u2.val ∈ Δ) :

    mul_M_raw Δ h_max h_con t1 t2 = mul_M_raw Δ h_max h_con u1 u2 := by

  unfold mul_M_raw

  apply Quotient.sound
```

```
  exact mul_congruence Δ h_max h_con t1 t2 u1 u2 h1 h2


def mul_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ)
(x y : M Δ h_max h_con) : M Δ h_max h_con :=

  Quotient.lift (fun (t1 : TermClosed s) =>

    Quotient.lift (fun (t2 : TermClosed s) => mul_M_raw Δ h_max h_con t1 t2)

     (fun t2 u2 h2 => mul_M_raw_congruence Δ h_max h_con t1 t2 t1 u2
(maximal_consistent_closed_under_deduction Δ h_max h_con _
(SPA_Deduction_Internal.eq_refl Δ t1.val)) h2) y)

    (fun t1 u1 h1 => by

      induction y using Quotient.inductionOn

      rename_i t2

      dsimp

      exact mul_M_raw_congruence Δ h_max h_con t1 t2 u1 t2 h1
(maximal_consistent_closed_under_deduction Δ h_max h_con _
(SPA_Deduction_Internal.eq_refl Δ t2.val))) x


def exp_M_raw {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ) (t1 t2 : TermClosed s) : M Δ h_max h_con :=

  Quotient.mk (termSetoid Δ h_max h_con) (exp_closed t1 t2)


lemma exp_M_raw_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)

    (t1 t2 u1 u2 : TermClosed s) (h1 : ArithForm.eq t1.val u1.val ∈ Δ) (h2 : ArithForm.eq
t2.val u2.val ∈ Δ) :

    exp_M_raw Δ h_max h_con t1 t2 = exp_M_raw Δ h_max h_con u1 u2 := by

  unfold exp_M_raw

  apply Quotient.sound

  exact exp_congruence Δ h_max h_con t1 t2 u1 u2 h1 h2


def exp_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ)
(x y : M Δ h_max h_con) : M Δ h_max h_con :=
```

```
  Quotient.lift (fun (t1 : TermClosed s) =>

    Quotient.lift (fun (t2 : TermClosed s) => exp_M_raw Δ h_max h_con t1 t2)

      (fun t2 u2 h2 => exp_M_raw_congruence Δ h_max h_con t1 t2 t1 u2
(maximal_consistent_closed_under_deduction Δ h_max h_con _
(SPA_Deduction_Internal.eq_refl Δ t1.val)) h2) y)

    (fun t1 u1 h1 => by

      induction y using Quotient.inductionOn

      rename_i t2

      dsimp

      exact exp_M_raw_congruence Δ h_max h_con t1 t2 u1 t2 h1
(maximal_consistent_closed_under_deduction Δ h_max h_con _
(SPA_Deduction_Internal.eq_refl Δ t2.val))) x
```

Interpretaciones de los predicados en el modelo canónico

Se definen las interpretaciones de los predicados del lenguaje (los operadores estratificados `provable_op` y `true_op`, y la relación ternaria `list_nth_eq`) en el modelo canónico. Para cada predicado, se demuestra primero un lema de congruencia que garantiza que su valor de verdad en Δ es independiente del representante elegido, y luego se levanta al cociente mediante `Quotient.lift`.

Interpretación de `provable_op`

`provable_op_congruence`: Establece que el valor de verdad de `provable_op` en Δ es invariante bajo la relación de equivalencia de términos: si $t_1 \sim t_2$ (es decir, $(t_1 = t_2) \in \Delta$), entonces:

$$ \text{provable_op}_{h}(t_1) \in \Delta \iff \text{provable_op}_{h}(t_2) \in \Delta $$

La demostración utiliza la regla `eq_provable_op` del sistema deductivo en ambas direcciones, junto con simetría de la igualdad para la dirección inversa.

`provable_op_M`: Interpretación de `provable_op` en el modelo, definida por levantamiento:

$$ \text{provable_op}^M_{h}([t]_\sim) := (\text{provable_op}_{h}(t) \in \Delta) $$

El valor de verdad es una proposición (`Prop`) que es verdadera exactamente cuando la fórmula correspondiente pertenece a Δ. La buena definición está garantizada por `provable_op_congruence`, utilizando `propext` para convertir la equivalencia lógica en igualdad de proposiciones.

Interpretación de `true_op`

`true_op_congruence`: Análogo para el operador de verdad:

$$ t_1 \sim t_2 \implies (\text{true_op}_{h}(t_1) \in \Delta \iff \text{true_op}_{h}(t_2) \in \Delta) $$

La demostración es simétrica a la de `provable_op_congruence`, utilizando `eq_true_op`.

`true_op_M`: Interpretación de `true_op` en el modelo:

$$ \text{true_op}^M_{h}([t]_\sim) := (\text{true_op}_{h}(t) \in \Delta) $$

Congruencia para `list_nth_eq` (primer argumento)

`list_nth_eq_congruence_1`: Establece la invarianza bajo equivalencia en el primer argumento de la relación ternaria:

$$ t_1 \sim u_1 \implies (\text{list_nth_eq}(t_1, t_2, t_3) \in \Delta \iff \text{list_nth_eq}(u_1, t_2, t_3) \in \Delta) $$

La demostración utiliza la regla `eq_list_nth_eq_1` del sistema deductivo en ambas direcciones. Se requerirían lemas análogos para el segundo y tercer argumento (`list_nth_eq_congruence_2` y `list_nth_eq_congruence_3`) para completar la interpretación de la relación ternaria en el modelo.

Significado

Estas definiciones completan la interpretación de los predicados del lenguaje en el modelo canónico. El patrón es uniforme:

1. Se demuestra que el predicado es congruente con la relación de equivalencia de términos (es decir, que su valor de verdad en Δ no depende del representante elegido).

2. Se levanta el predicado al cociente usando `Quotient.lift`, donde el valor de verdad sobre una clase $[t]_\sim$ es simplemente la pertenencia de la fórmula correspondiente a Δ.

La igualdad en el modelo se interpreta de manera natural: dos clases $[t_1]_\sim$ y $[t_2]_\sim$ son iguales en el modelo exactamente cuando $(t_1 = t_2) \in \Delta$. Pero por definición de la relación de equivalencia `termSetoid`, esto ocurre exactamente cuando $[t_1]_\sim = [t_2]_\sim$ como elementos del cociente. Es decir, la igualdad del modelo coincide con la igualdad del cociente, por lo que no requiere una definición adicional.

Con estas interpretaciones, el modelo canónico $M(\Delta)$ queda completamente definido como una estructura de primer orden para el lenguaje de la aritmética estratificada SPA. El dominio es el cociente de términos cerrados, las funciones se interpretan sintácticamente, y los predicados se interpretan según la pertenencia a Δ. Esta construcción establece el escenario para la demostración del Teorema de Verdad (o Lema de Satisfacción), que afirma que para toda fórmula φ, $M \models \varphi$ si y solo si $\varphi \in \Delta$. Este teorema completa la prueba de completitud: como Δ extiende a la teoría original consistente Γ, el modelo $M(\Delta)$ satisface Γ, demostrando que toda teoría consistente tiene un modelo. Se han completado las interpretaciones de los predicados en el modelo canónico. Se definieron los lemas de congruencia y las interpretaciones para `provable_op`, `true_op` y `list_nth_eq` (primer argumento), siguiendo el patrón de levantamiento al cociente mediante `Quotient.lift`. La estructura del modelo queda completamente definida con dominio en las clases de equivalencia de términos cerrados, funciones interpretadas sintácticamente y predicados interpretados según pertenencia a Δ, estableciendo las bases para el Teorema de Verdad que completaría la prueba de completitud.

```
--
================================================================

-- INTERPRETACIONES DE LOS PREDICADOS EN EL MODELO CANÓNICO

--
================================================================


lemma provable_op_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)

  {s_l : Nat} (h_eq : s = s_l + 1) (t1 t2 : TermClosed s) (h : ArithForm.eq t1.val t2.val ∈
Δ) :

  (ArithForm.provable_op h_eq t1.val ∈ Δ) ↔ (ArithForm.provable_op h_eq t2.val ∈ Δ)
:= by

 constructor

 · intro h1

  apply maximal_consistent_closed_under_deduction Δ h_max h_con

  apply SPA_Deduction_Internal.eq_provable_op _ h_eq t1.val t2.val

  · exact SPA_Deduction_Internal.hyp Δ _ h

  · exact SPA_Deduction_Internal.hyp Δ _ h1

 · intro h2

  apply maximal_consistent_closed_under_deduction Δ h_max h_con

  apply SPA_Deduction_Internal.eq_provable_op _ h_eq t2.val t1.val

  · apply SPA_Deduction_Internal.eq_symm

   exact SPA_Deduction_Internal.hyp Δ _ h

  · exact SPA_Deduction_Internal.hyp Δ _ h2
```

```
def provable_op_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ)
  {s_l : Nat} (h_eq : s = s_l + 1) (x : M Δ h_max h_con) : Prop :=
 Quotient.lift (fun (t : TermClosed s) => ArithForm.provable_op h_eq t.val ∈ Δ)
  (fun t1 t2 h => propext (provable_op_congruence Δ h_max h_con h_eq t1 t2 h)) x

lemma true_op_congruence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)
  {s_l : Nat} (h_eq : s = s_l + 1) (t1 t2 : TermClosed s) (h : ArithForm.eq t1.val t2.val ∈ Δ) :
  (ArithForm.true_op h_eq t1.val ∈ Δ) ↔ (ArithForm.true_op h_eq t2.val ∈ Δ) := by
 constructor
 · intro h1
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.eq_true_op _ h_eq t1.val t2.val
  · exact SPA_Deduction_Internal.hyp Δ _ h
  · exact SPA_Deduction_Internal.hyp Δ _ h1
 · intro h2
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.eq_true_op _ h_eq t2.val t1.val
  · apply SPA_Deduction_Internal.eq_symm
    exact SPA_Deduction_Internal.hyp Δ _ h
  · exact SPA_Deduction_Internal.hyp Δ _ h2

def true_op_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ)
  {s_l : Nat} (h_eq : s = s_l + 1) (x : M Δ h_max h_con) : Prop :=
 Quotient.lift (fun (t : TermClosed s) => ArithForm.true_op h_eq t.val ∈ Δ)
  (fun t1 t2 h => propext (true_op_congruence Δ h_max h_con h_eq t1 t2 h)) x
```

```
lemma list_nth_eq_congruence_1 {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)

  (t1 t2 t3 u1 : TermClosed s) (h : ArithForm.eq t1.val u1.val ∈ Δ) :

  (ArithForm.list_nth_eq t1.val t2.val t3.val ∈ Δ) ↔ (ArithForm.list_nth_eq u1.val t2.val
t3.val ∈ Δ) := by

 constructor

 · intro h1

  apply maximal_consistent_closed_under_deduction Δ h_max h_con

  apply SPA_Deduction_Internal.eq_list_nth_eq_1 _ _ _ _ _

  · exact SPA_Deduction_Internal.hyp Δ _ h

  · exact SPA_Deduction_Internal.hyp Δ _ h1

 · intro h2

  apply maximal_consistent_closed_under_deduction Δ h_max h_con

  apply SPA_Deduction_Internal.eq_list_nth_eq_1 _ _ _ _ _

  · apply SPA_Deduction_Internal.eq_symm

   exact SPA_Deduction_Internal.hyp Δ _ h

  · exact SPA_Deduction_Internal.hyp Δ _ h2
```

Congruencia completa para `list_nth_eq` y su interpretación en el modelo canónico

Se completan los lemas de congruencia para los tres argumentos de la relación ternaria `list_nth_eq` y se define su interpretación en el modelo canónico mediante levantamiento anidado triple.

Lemas de congruencia para los argumentos restantes

`list_nth_eq_congruence_2`: Invarianza bajo equivalencia en el segundo argumento:

$$ t_2 \sim u_2 \implies (\text{list_nth_eq}(t_1, t_2, t_3) \in \Delta \iff \text{list_nth_eq}(t_1, u_2, t_3) \in \Delta) $$

Utiliza la regla `eq_list_nth_eq_2` del sistema deductivo en ambas direcciones.

`list_nth_eq_congruence_3`: Invarianza bajo equivalencia en el tercer argumento:

$$ t_3 \sim u_3 \implies (\text{list_nth_eq}(t_1, t_2, t_3) \in \Delta \iff \text{list_nth_eq}(t_1, t_2, u_3) \in \Delta) $$

Utiliza la regla `eq_list_nth_eq_3` del sistema deductivo en ambas direcciones.

Interpretación `list_nth_eq_M` en el modelo canónico

Define la relación ternaria en el modelo mediante tres levantamientos anidados:

$$ \text{list_nth_eq}^M([t_1]_\sim, [t_2]_\sim, [t_3]_\sim) := (\text{list_nth_eq}(t_1, t_2, t_3) \in \Delta) $$

La construcción procede de adentro hacia afuera:

1. **Levantamiento sobre el tercer argumento:** Se define una función que toma t_1, t_2 fijos y levanta sobre x_3 usando `list_nth_eq_congruence_3`.

2. **Levantamiento sobre el segundo argumento:** Se toma t_1 fijo y se levanta sobre x_2. La prueba de congruencia requiere inducción sobre el representante de x_3 (obtenido mediante `Quotient.inductionOn`) y utiliza `list_nth_eq_congruence_2`.

3. **Levantamiento sobre el primer argumento:** Finalmente se levanta sobre x_1. La prueba de congruencia requiere inducción sobre los representantes de x_2 y x_3, y utiliza `list_nth_eq_congruence_1`.

En cada paso se utiliza `propext` para convertir la equivalencia lógica de proposiciones en igualdad, como requiere `Quotient.lift`.

Significado

Con estas definiciones, el modelo canónico $M(\Delta)$ queda completamente equipado con interpretaciones para todos los símbolos del lenguaje `ArithForm`:

| Símbolo | Tipo | Interpretación |

|---------|------|---------------|

| `zero` | constante | $[0]_\sim$ |

| `const c` | constante | $[c]_\sim$ |

| `henkin c` | constante | $[\text{henkin } c]_\sim$ |

| `succ` | función unaria | $S^M([t]_\sim) = [S(t)]_\sim$ |

| `pred` | función unaria | $\text{pred}^M([t]_\sim) = [\text{pred}(t)]_\sim$ |

| `add` | función binaria | $[t_1]_\sim +^M [t_2]_\sim = [t_1 + t_2]_\sim$ |

| `mul` | función binaria | $[t_1]_\sim \cdot^M [t_2]_\sim = [t_1 \cdot t_2]_\sim$ |

| `exp` | función binaria | $[t_1]_\sim \text{ exp}^M [t_2]_\sim = [t_1^{t_2}]_\sim$ |

| `eq` | relación binaria | Igualdad del cociente |

| `provable_op` | predicado unario | Verdad si la fórmula está en Δ |

| `true_op` | predicado unario | Verdad si la fórmula está en Δ |

| `list_nth_eq` | relación ternaria | Verdad si la fórmula está en Δ |

La construcción sistemática de interpretaciones mediante `Quotient.lift` garantiza que todas las operaciones y relaciones están bien definidas sobre las clases de equivalencia. El patrón de levantamiento anidado para funciones y relaciones de aridad superior es completamente general y podría aplicarse a cualquier signatura.

Este modelo canónico, conocido como modelo de Henkin, tiene la propiedad fundamental de que la verdad en el modelo coincide con la pertenencia a Δ: para toda fórmula $\varphi(\vec{x})$ y toda asignación de términos cerrados $\vec{t}$,

$$ M \models \varphi([\vec{t}]_\sim) \iff \varphi(\vec{t}) \in \Delta $$

Este es el Teorema de Verdad (o Lema de Satisfacción) que constituye el núcleo de la prueba de completitud. Su demostración procede por inducción sobre la complejidad de φ, utilizando la maximalidad de Δ para los casos de negación e implicación, y la propiedad de Henkin para el caso del cuantificador existencial. Con

esto, toda teoría consistente Γ tiene un modelo (el modelo canónico de su extensión de Henkin-Lindenbaum Δ), completando así el Teorema de Completitud de Gödel para el sistema SPA.

lemma list_nth_eq_congruence_2 {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

(t1 t2 t3 u2 : TermClosed s) (h : ArithForm.eq t2.val u2.val ∈ Δ) :

(ArithForm.list_nth_eq t1.val t2.val t3.val ∈ Δ) ↔ (ArithForm.list_nth_eq t1.val u2.val t3.val ∈ Δ) := by

constructor

· intro h1

apply maximal_consistent_closed_under_deduction Δ h_max h_con

apply SPA_Deduction_Internal.eq_list_nth_eq_2 _ _ _ _ _

· exact SPA_Deduction_Internal.hyp Δ _ h

· exact SPA_Deduction_Internal.hyp Δ _ h1

· intro h2

apply maximal_consistent_closed_under_deduction Δ h_max h_con

apply SPA_Deduction_Internal.eq_list_nth_eq_2 _ _ _ _ _

· apply SPA_Deduction_Internal.eq_symm

```
    exact SPA_Deduction_Internal.hyp Δ _ h
  · exact SPA_Deduction_Internal.hyp Δ _ h2


lemma list_nth_eq_congruence_3 {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)
  (t1 t2 t3 u3 : TermClosed s) (h : ArithForm.eq t3.val u3.val ∈ Δ) :
  (ArithForm.list_nth_eq t1.val t2.val t3.val ∈ Δ) ↔ (ArithForm.list_nth_eq t1.val t2.val
u3.val ∈ Δ) := by
 constructor
 · intro h1
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.eq_list_nth_eq_3 _ _ _ _ _
  · exact SPA_Deduction_Internal.hyp Δ _ h
  · exact SPA_Deduction_Internal.hyp Δ _ h1
 · intro h2
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.eq_list_nth_eq_3 _ _ _ _ _
  · apply SPA_Deduction_Internal.eq_symm
    exact SPA_Deduction_Internal.hyp Δ _ h
  · exact SPA_Deduction_Internal.hyp Δ _ h2


def list_nth_eq_M {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con :
Consistent Δ)
  (x1 x2 x3 : M Δ h_max h_con) : Prop :=
 Quotient.lift (fun (t1 : TermClosed s) =>
  Quotient.lift (fun (t2 : TermClosed s) =>
   Quotient.lift (fun (t3 : TermClosed s) => ArithForm.list_nth_eq t1.val t2.val t3.val ∈
Δ)
    (fun t3 u3 h3 => propext (list_nth_eq_congruence_3 Δ h_max h_con t1 t2 t3 u3 h3))
x3)
```

```
    (fun t2 u2 h2 => propext (by
      induction x3 using Quotient.inductionOn
      rename_i t3
      dsimp
      exact list_nth_eq_congruence_2 Δ h_max h_con t1 t2 t3 u2 h2)) x2)
  (fun t1 u1 h1 => propext (by
    induction x2 using Quotient.inductionOn
    rename_i t2
    induction x3 using Quotient.inductionOn
    rename_i t3
    dsimp
    exact list_nth_eq_congruence_1 Δ h_max h_con t1 t2 t3 u1 h1)) x1
```

Estructura de modelo y construcción del modelo canónico

Se define la estructura general de un modelo para el lenguaje de la aritmética estratificada SPA, y se construye el modelo canónico a partir de una teoría maximal consistente con propiedad de Henkin.

Definición `Model s`: Estructura de modelo para el estrato s

Define un registro que agrupa todos los componentes necesarios para interpretar el lenguaje `ArithForm s`:

$$ \text{Model } s := \begin{cases}

\text{Carrier} &: \text{Type} \quad \text{(dominio)} \\

\text{zero} &: \text{Carrier} \\

\text{succ} &: \text{Carrier} \to \text{Carrier} \\

\text{add} &: \text{Carrier} \to \text{Carrier} \to \text{Carrier} \\

\text{mul} &: \text{Carrier} \to \text{Carrier} \to \text{Carrier} \\

\text{exp} &: \text{Carrier} \to \text{Carrier} \to \text{Carrier} \\

\text{pred} &: \text{Carrier} \to \text{Carrier} \\

\text{const} &: \text{NatS } s \to \text{Carrier} \\

\text{henkin} &: \text{HenkinConst} \to \text{Carrier} \\

\text{eq_rel} &: \text{Carrier} \to \text{Carrier} \to \text{Prop} \\

\text{provable_op_rel} &: \{s_{\text{lower}}\} \to (s = s_{\text{lower}} + 1) \to \text{Carrier} \to \text{Prop} \\

\text{true_op_rel} &: \{s_{\text{lower}}\} \to (s = s_{\text{lower}} + 1) \to \text{Carrier} \to \text{Prop} \\

\text{list_nth_eq_rel} &: \text{Carrier} \to \text{Carrier} \to \text{Carrier} \to \text{Prop}

\end{cases} $$

Cada componente corresponde exactamente a un símbolo del lenguaje. Los operadores estratificados (`provable_op_rel`, `true_op_rel`) incluyen el testigo de igualdad de estratos como parámetro adicional.

Construcción `model_from_maximal`: Modelo canónico a partir de Δ

Construye el modelo canónico a partir de una teoría maximal consistente Δ con propiedad de Henkin:

$$ \text{model_from_maximal } \Delta\ h_{\text{max}}\ h_{\text{henkin}}\ h_{\text{con}} : \text{Model } s $$

Las interpretaciones se definen componente por componente:

Componente del modelo	Interpretación
`Carrier`	$M(\Delta, h_{\text{max}}, h_{\text{con}})$ (cociente de términos cerrados)
`zero`	`zero_M` (clase de `zero`)
`succ`	`succ_M` (sucesor sintáctico elevado)
`add`	`add_M` (adición sintáctica elevada)
`mul`	`mul_M` (multiplicación sintáctica elevada)
`exp`	`exp_M` (exponenciación sintáctica elevada)
`pred`	`pred_M` (predecesor sintáctico elevado)
`const`	`const_M` (constante numérica elevada)
`henkin`	`henkin_M` (constante de Henkin elevada)
`eq_rel`	Igualdad del cociente (dos clases son iguales si y solo si son la misma clase)
`provable_op_rel`	`provable_op_M` (verdad si la fórmula está en Δ)
`true_op_rel`	`true_op_M` (verdad si la fórmula está en Δ)
`list_nth_eq_rel`	`list_nth_eq_M` (verdad si la fórmula está en Δ)

Significado

Con esta construcción se completa la definición del modelo canónico para el sistema SPA. La estructura `Model s` captura la signatura completa del lenguaje, y `model_from_maximal` proporciona una implementación concreta donde:

- El dominio es el conjunto de clases de equivalencia de términos cerrados bajo la igualdad demostrable en Δ.

- Las funciones se interpretan sintácticamente: aplicar una función a clases de equivalencia consiste en aplicar la función sintáctica a representantes y tomar la clase del resultado.

- Las relaciones se interpretan según la pertenencia a Δ: una fórmula atómica es verdadera en el modelo exactamente cuando pertenece a la teoría maximal Δ.

- La igualdad en el modelo coincide con la identidad de clases de equivalencia, que por definición equivale a la igualdad demostrable en Δ.

Esta construcción es la culminación del método de Henkin-Lindenbaum para la prueba de completitud. El modelo canónico tiene la propiedad fundamental de que la verdad coincide con la pertenencia a Δ:

$$ M \models \varphi \iff \varphi \in \Delta $$

Esta propiedad, conocida como el Lema de Verdad o Lema de Satisfacción, se demuestra por inducción sobre la estructura de φ. Los casos atómicos se siguen directamente de las definiciones de las interpretaciones; los casos de conectivas proposicionales usan la maximalidad de Δ; el caso del cuantificador existencial usa la propiedad de Henkin; y el caso del cuantificador universal se reduce al existencial por dualidad.

Como Δ es una extensión de la teoría original consistente Γ, el modelo canónico satisface Γ, lo que demuestra que toda teoría consistente tiene un modelo. Este es el Teorema de Completitud de Gödel para el sistema estratificado SPA. La construcción `model_from_maximal` encapsula todo el proceso: dada una teoría consistente, se la extiende a una teoría de Henkin (por

`GammaLimit`), luego a una maximal consistente (por `LindenbaumLimit`), y finalmente se construye el modelo canónico a partir de ella.

```
--
============================================================================
-- ESTRUCTURA DE MODELO Y CONSTRUCCIÓN DEL MODELO CANÓNICO
--
============================================================================

structure Model (s : Nat) where
  Carrier : Type
  zero : Carrier
  succ : Carrier → Carrier
  add : Carrier → Carrier → Carrier
  mul : Carrier → Carrier → Carrier
  exp : Carrier → Carrier → Carrier
  pred : Carrier → Carrier
  const : NatS s → Carrier
  henkin : HenkinConst → Carrier
  eq_rel : Carrier → Carrier → Prop
```

```
  provable_op_rel : {s_lower : Nat} → (s = s_lower + 1) → Carrier → Prop
  true_op_rel : {s_lower : Nat} → (s = s_lower + 1) → Carrier → Prop
  list_nth_eq_rel : Carrier → Carrier → Carrier → Prop

def model_from_maximal {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (_h_henkin : HenkinSet Δ) (h_con : Consistent Δ) : Model s where
  Carrier := M Δ h_max h_con
  zero := zero_M h_max h_con
  succ := succ_M h_max h_con
  add := add_M h_max h_con
  mul := mul_M h_max h_con
  exp := exp_M h_max h_con
  pred := pred_M h_max h_con
  const := const_M h_max h_con
  henkin := henkin_M h_max h_con
  eq_rel := fun x y => x = y
  provable_op_rel := fun h_eq x => provable_op_M h_max h_con h_eq x
  true_op_rel := fun h_eq x => true_op_M h_max h_con h_eq x
  list_nth_eq_rel := fun x y z => list_nth_eq_M h_max h_con x y z

end SPA
```

Funciones de grounding y sustitución global

Se definen las operaciones de "grounding" (reemplazo de todas las variables por una constante de Henkin) y de sustitución global (reemplazo simultáneo de todas las variables según un entorno), junto con la propiedad de que la sustitución global preserva la clausura de términos.

Funciones `groundTerm` y `groundForm`: Grounding con constante de Henkin

Reemplazan todas las ocurrencias de variables en un término o fórmula por una constante de Henkin fija c:

$$ \text{groundTerm}(t, c) := t[\text{toda variable} := \text{henkin } c] $$

$$ \text{groundForm}(\varphi, c) := \varphi[\text{toda variable} := \text{henkin } c] $$

La definición es recursiva estructural:

- **`var n`:** Se reemplaza por `henkin c`.

- **Constantes, Henkin, zero:** Se preservan.

- **Constructores compuestos:** La operación se propaga a los subtérminos/subfórmulas.

- **`forall_form n f`:** El cuantificador se preserva y la sustitución se aplica al cuerpo. Esto significa que las variables ligadas no son reemplazadas (la sustitución no penetra el cuantificador).

Intuición y necesidad

El grounding es una operación técnica utilizada en ciertas demostraciones de completitud donde se necesita transformar fórmulas con variables libres en fórmulas

cerradas (sin variables), reemplazando todas las variables por una constante de Henkin fija. Esto permite aplicar la propiedad de Henkin (que solo se aplica a fórmulas cerradas) a fórmulas que originalmente tenían variables libres.

Funciones `substAllTerm`, `substAll_raw` y `substAll`: Sustitución global

`substAllTerm t env`: Reemplaza simultáneamente cada variable v en t por el término `env v`:

$$ \text{substAllTerm}(t, \text{env}) := t[v_0 := \text{env}(0), v_1 := \text{env}(1), \ldots] $$

`substAll_raw φ env`: Análogo para fórmulas. En el caso de `forall_form n f`, el entorno se modifica para que n se mapee a $\text{var } n$ (la variable ligada no debe ser sustituida):

$$ \text{substAll_raw}(\forall n\ f, \text{env}) := \forall n\ (\text{substAll_raw}(f, \text{env}[n := \text{var } n])) $$

`substAll φ env`: Versión que toma un entorno de términos cerrados (`TermClosed s`) y aplica `substAll_raw` con los valores subyacentes.

Lema `isClosed_substAllTerm`: Clausura bajo sustitución global

Establece que si todas las variables se sustituyen por términos cerrados, el resultado es un término cerrado:

$$ \forall t \in \text{ArithTerm } s,\ \forall \text{env} : \mathbb{N} \to \text{TermClosed } s,\ \text{isClosed}(\text{substAllTerm } t\ (\lambda v.\ (\text{env } v).\text{val})) = \text{true} $$

La demostración es por inducción sobre t:

- **`var n`:** Se reemplaza por `(env n).val`, que es cerrado por la propiedad `(env n).property`.

- **Casos base (`const`, `henkin`, `zero`):** Ya son cerrados.

- **Casos recursivos:** La clausura se preserva al aplicar constructores a términos cerrados. Para `add`, `mul`, `exp`, se utiliza la conjunción de las clausuras de los subtérminos.

Significado

Estas definiciones proporcionan las operaciones fundamentales de manipulación de variables en el lenguaje. La sustitución global `substAll` es una generalización de la sustitución simple `substForm` que permite reemplazar todas las variables libres simultáneamente, lo cual es útil para definir la noción de satisfacción de una fórmula en un modelo bajo una asignación de variables.

En la demostración del Teorema de Verdad para el modelo canónico, la satisfacción de una fórmula φ bajo una asignación $\sigma : \mathbb{N} \to M(\Delta)$ se definirá como:

$$ M \models_\sigma \varphi \iff \text{substAll } \varphi\ \sigma' \in \Delta $$

donde σ' es un entorno de términos cerrados que levanta σ eligiendo representantes para cada clase de equivalencia. El lema `isClosed_substAllTerm` garantiza que la fórmula resultante es cerrada (sin variables libres), lo cual es necesario para que la pertenencia a Δ esté bien definida (ya que Δ es una teoría sobre fórmulas del lenguaje, y las variables libres requerirían una noción

de satisfacción bajo asignación, mientras que las fórmulas cerradas tienen un valor de verdad absoluto en Δ).

namespace SPA

```
--
================================================================

-- FUNCIONES DE GROUNDING: REEMPLAZAN VARIABLES POR CONSTANTE DE
HENKIN

--
================================================================


def groundTerm {s : Nat} (t : ArithTerm s) (c : HenkinConst) : ArithTerm s :=

  match t with

  | ArithTerm.var _ => ArithTerm.henkin c

  | ArithTerm.const c' => ArithTerm.const c'

  | ArithTerm.henkin c' => ArithTerm.henkin c'

  | ArithTerm.zero => ArithTerm.zero

  | ArithTerm.succ t' => ArithTerm.succ (groundTerm t' c)

  | ArithTerm.add t1 t2 => ArithTerm.add (groundTerm t1 c) (groundTerm t2 c)

  | ArithTerm.mul t1 t2 => ArithTerm.mul (groundTerm t1 c) (groundTerm t2 c)
```

```
  | ArithTerm.exp t1 t2 => ArithTerm.exp (groundTerm t1 c) (groundTerm t2 c)
  | ArithTerm.pred t' => ArithTerm.pred (groundTerm t' c)

def groundForm {s : Nat} (φ : ArithForm s) (c : HenkinConst) : ArithForm s :=
  match φ with
  | ArithForm.bottom => ArithForm.bottom
  | ArithForm.eq t1 t2 => ArithForm.eq (groundTerm t1 c) (groundTerm t2 c)
  | ArithForm.impl f1 f2 => ArithForm.impl (groundForm f1 c) (groundForm f2 c)
  | ArithForm.neg f => ArithForm.neg (groundForm f c)
  | ArithForm.forall_form n f => ArithForm.forall_form n (groundForm f c)
  | ArithForm.provable_op h t => ArithForm.provable_op h (groundTerm t c)
  | ArithForm.true_op h t => ArithForm.true_op h (groundTerm t c)
  | ArithForm.list_nth_eq t1 t2 t3 => ArithForm.list_nth_eq (groundTerm t1 c)
(groundTerm t2 c) (groundTerm t3 c)



--
============================================================================
-- DEFINICIÓN DE SUSTITUCIÓN GLOBAL Y PROPIEDADES DE CERRADURA
--
============================================================================

def substAllTerm {s : Nat} (t : ArithTerm s) (env : Nat → ArithTerm s) : ArithTerm s :=
  match t with
  | ArithTerm.var n => env n
  | ArithTerm.const c => ArithTerm.const c
  | ArithTerm.henkin c => ArithTerm.henkin c
  | ArithTerm.zero => ArithTerm.zero
  | ArithTerm.succ t' => ArithTerm.succ (substAllTerm t' env)
```

```
  | ArithTerm.add t1 t2 => ArithTerm.add (substAllTerm t1 env) (substAllTerm t2 env)
  | ArithTerm.mul t1 t2 => ArithTerm.mul (substAllTerm t1 env) (substAllTerm t2 env)
  | ArithTerm.exp t1 t2 => ArithTerm.exp (substAllTerm t1 env) (substAllTerm t2 env)
  | ArithTerm.pred t' => ArithTerm.pred (substAllTerm t' env)

def substAll_raw {s : Nat} (φ : ArithForm s) (env : Nat → ArithTerm s) : ArithForm s :=
  match φ with
  | ArithForm.bottom => ArithForm.bottom
  | ArithForm.eq t1 t2 => ArithForm.eq (substAllTerm t1 env) (substAllTerm t2 env)
  | ArithForm.impl f1 f2 => ArithForm.impl (substAll_raw f1 env) (substAll_raw f2 env)
  | ArithForm.neg f => ArithForm.neg (substAll_raw f env)
  | ArithForm.forall_form n f =>
    ArithForm.forall_form n (substAll_raw f (fun v => if v = n then ArithTerm.var n else env v))
  | ArithForm.provable_op h t => ArithForm.provable_op h (substAllTerm t env)
  | ArithForm.true_op h t => ArithForm.true_op h (substAllTerm t env)
  | ArithForm.list_nth_eq t1 t2 t3 => ArithForm.list_nth_eq (substAllTerm t1 env) (substAllTerm t2 env) (substAllTerm t3 env)

def substAll {s : Nat} (φ : ArithForm s) (env : Nat → TermClosed s) : ArithForm s :=
  substAll_raw φ (fun v => (env v).val)

lemma isClosed_substAllTerm {s : Nat} (t : ArithTerm s) (env : Nat → TermClosed s) :
    isClosed (substAllTerm t (fun v => (env v).val)) = true := by
  induction t with
  | var n => exact (env n).property
  | const c => rfl
  | henkin c => rfl
```

```
| zero => rfl
| succ t' ih => exact ih
| add t1 t2 ih1 ih2 =>
  dsimp [substAllTerm, isClosed]
  rw [ih1, ih2]
  rfl
| mul t1 t2 ih1 ih2 =>
  dsimp [substAllTerm, isClosed]
  rw [ih1, ih2]
  rfl
| exp t1 t2 ih1 ih2 =>
  dsimp [substAllTerm, isClosed]
  rw [ih1, ih2]
  rfl
| pred t' ih => exact ih
```

Definición de `evalTerm` y lemas de clausura

Se define la función de evaluación de términos cerrados en el modelo canónico y se demuestran dos lemas fundamentales sobre la relación entre clausura y las propiedades sintácticas de no ocurrencia y libertad.

Función `evalTerm`: Evaluación de términos cerrados en el modelo

Define la interpretación de un término cerrado t en el modelo canónico como su clase de equivalencia:

$$ \text{evalTerm } \Delta\ h_{\text{max}}\ h_{\text{con}}\ t\ h_{\text{closed}} := [t]_\sim \in M(\Delta) $$

El argumento `h_closed` es una prueba de que `isClosed t = true`, necesaria para construir el elemento del subtipo `TermClosed s` que requiere `Quotient.mk`.

Intuición

Esta función establece el puente entre la sintaxis (términos cerrados) y la semántica (elementos del modelo). Un término cerrado se interpreta como su propia clase de equivalencia en el cociente. Esta es la definición estándar en el modelo de Henkin: los términos cerrados se interpretan "como ellos mismos" módulo la igualdad demostrable.

Lema `NotOccur_of_isClosed`: Los términos cerrados no contienen variables

Establece formalmente que si un término es cerrado, entonces ninguna variable ocurre en él:

$$ \forall t \in \text{ArithTerm } s,\ \text{isClosed } t = \text{true} \implies \forall w \in \mathbb{N},\ \text{NotOccur } w\ t $$

Estructura de la demostración

Por inducción sobre t:

- **`var n`:** `isClosed (var n) = false`, lo que contradice la hipótesis. Por tanto, este caso es imposible (un término cerrado no puede ser una variable).

- **`const`, `henkin`, `zero`:** Son constructores que no contienen variables, por lo que `NotOccur w` se cumple para cualquier w.

- **`succ t'`, `pred t'`:** La hipótesis `isClosed (succ t') = true` implica `isClosed t' = true`. Por hipótesis de inducción, `NotOccur w t'`, y por tanto `NotOccur w (succ t')`.

- **`add`, `mul`, `exp`:** La clausura de un término compuesto implica la clausura de ambos subtérminos (pues `isClosed` usa conjunción). Se aplican las hipótesis de inducción a cada uno y se combinan con el constructor correspondiente de `NotOccur`.

Significado

Este lema es fundamental para toda la maquinaria de sustitución y frescura. Garantiza que los términos cerrados son "inmunes" a las sustituciones: como ninguna variable ocurre en ellos, `substTerm v u t = t` para cualquier v y u. Esta propiedad se ha utilizado profusamente en las demostraciones de representabilidad mediante `substTerm_fresh`.

Lema `IsFreeFor_closed`: Los términos cerrados son libres para cualquier variable

Establece que cualquier término cerrado t es libre para cualquier variable v en cualquier fórmula φ:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \forall t \in \text{TermClosed } s,\ \text{IsFreeFor } v\ t.\text{val}\ \varphi $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos sin cuantificadores (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`):** `IsFreeFor` es trivialmente verdadero.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Si $v = n$, la sustitución no penetra y la condición es trivial. Si $v \neq n$, se requiere:

  - `NotOccur n t.val`: Como t es cerrado, `NotOccur_of_isClosed` garantiza que n no ocurre en t.

  - `IsFreeFor v t.val f`: Por hipótesis de inducción.

Significado

Este lema es crucial para la aplicación de la regla `forall_elim` en contextos donde se instancian cuantificadores con términos cerrados. Garantiza que nunca hay riesgo de captura de variables al sustituir un término cerrado, ya que los términos cerrados no contienen variables libres que pudieran ser capturadas por cuantificadores. Esta

propiedad es utilizada implícitamente en todas las demostraciones donde se aplica `forall_elim` con numerales o constantes de Henkin, y es la razón por la que las demostraciones de inducción interna pueden instanciar cuantificadores universales con términos cerrados sin verificar condiciones adicionales de libertad.

```
--
================================================================

-- 1. DEFINICIÓN DE evalTerm (PARÁMETROS DE MODELO EXPLICITADOS)

--
================================================================


def evalTerm {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent
Δ)

   (t : ArithTerm s) (h_closed : isClosed t = true) : M Δ h_max h_con :=

  Quotient.mk (termSetoid Δ h_max h_con) ⟨t, h_closed⟩


--
================================================================

-- 2. LEMA NotOccur_of_isClosed

--
================================================================
```

```
lemma NotOccur_of_isClosed {s : Nat} (t : ArithTerm s) (h_closed : isClosed t = true) (w
: Nat) : NotOccur w t := by
 induction t with
 | var n =>
  contradiction
 | const c =>
  exact NotOccur.const _
 | henkin c =>
  exact NotOccur.henkin _
 | zero =>
  exact NotOccur.zero
 | succ t' ih =>
  dsimp [isClosed] at h_closed
  exact NotOccur.succ _ (ih h_closed)
 | add t1 t2 ih1 ih2 =>
  dsimp [isClosed] at h_closed
  rw [Bool.and_eq_true] at h_closed
  exact NotOccur.add _ _ (ih1 h_closed.1) (ih2 h_closed.2)
 | mul t1 t2 ih1 ih2 =>
  dsimp [isClosed] at h_closed
  rw [Bool.and_eq_true] at h_closed
  exact NotOccur.mul _ _ (ih1 h_closed.1) (ih2 h_closed.2)
 | exp t1 t2 ih1 ih2 =>
  dsimp [isClosed] at h_closed
  rw [Bool.and_eq_true] at h_closed
  exact NotOccur.exp _ _ (ih1 h_closed.1) (ih2 h_closed.2)
 | pred t' ih =>
  dsimp [isClosed] at h_closed
```

```
  exact NotOccur.pred _ (ih h_closed)


--
============================================================================
-- 3. LEMA IsFreeFor_closed (RESOLUCIÓN DE METAVARIABLES DE GOAL)
--
============================================================================


lemma IsFreeFor_closed {s : Nat} (φ : ArithForm s) (v : Nat) (t : TermClosed s) :
IsFreeFor v t.val φ := by
 induction φ generalizing v with
 | bottom => trivial
 | eq t1 t2 => trivial
 | impl f1 f2 ih1 ih2 =>
  exact ⟨ih1 v, ih2 v⟩
 | neg f ih =>
  exact ih v
 | forall_form n f ih =>
  dsimp [IsFreeFor]
  by_cases h_vn : v = n
  · rw [if_pos h_vn]
   trivial
  · rw [if_neg h_vn]
   exact ⟨NotOccur_of_isClosed t.val t.property n, ih v⟩
 | provable_op _ _ => trivial
 | true_op _ _ => trivial
 | list_nth_eq _ _ _ => trivial
```

Definición de `satisfies` y lema de sustitución para `forall_form`

Se define la relación de satisfacción de una fórmula en el modelo canónico bajo una asignación de variables, y se demuestra un lema técnico sobre la conmutación de la sustitución con la función `substAllTerm` en el contexto de cuantificadores.

Función `satisfies`: Satisfacción en el modelo canónico

Define la relación $M \models_\sigma \varphi$ (la fórmula φ es satisfecha en el modelo canónico bajo la asignación σ) por recursión estructural sobre φ:

$$ \text{satisfies } \Delta\ h_{\text{max}}\ h_{\text{henkin}}\ h_{\text{con}}\ \sigma\ \varphi : \text{Prop} $$

donde $\sigma : \mathbb{N} \to \text{TermClosed } s$ asigna a cada variable un término cerrado. Las cláusulas son:

- **`bottom`:** Falso (nunca se satisface).

- **`eq t1 t2`:** La igualdad se satisface si las evaluaciones de t_1 y t_2 (tras sustituir las variables según σ) son iguales en M. La clausura de los términos sustituidos está garantizada por `isClosed_substAllTerm`.

- **`impl f1 f2`:** La implicación se interpreta como implicación en el metalenguaje.

- **`neg f`:** La negación se interpreta como negación en el metalenguaje.

- **`forall_form n f`:** La fórmula $\forall n\ f$ se satisface si para todo término cerrado t, f se satisface bajo la asignación modificada $\sigma[n := t]$.

- **`provable_op heq t`:** Se satisface si `provable_op_M` es verdadero para la evaluación de t.

- **`true_op heq t`:** Se satisface si `true_op_M` es verdadero para la evaluación de t.

- **`list_nth_eq t1 t2 t3`:** Se satisface si `list_nth_eq_M` es verdadero para las evaluaciones de los tres términos.

Lema `substTerm_substAllTerm_forall_form_general`: Conmutación para `forall_form`

Establece una propiedad fundamental de conmutación entre la sustitución simple y la sustitución global en el contexto de cuantificadores. Si n no ocurre en ninguna imagen del entorno `env`, entonces:

$$ \text{substTerm } n\ u\ (\text{substAllTerm } t\ (\lambda v.\ \text{if } v = n \text{ then var } n \text{ else env } v)) = \text{substAllTerm } t\ (\lambda v.\ \text{if } v = n \text{ then } u \text{ else env } v) $$

Estructura de la demostración

Por inducción sobre t:

- **`var m`:** Si $m = n$, el lado izquierdo sustituye n por u en $\text{var } n$, resultando u. El lado derecho aplica el nuevo entorno a m, que como $m = n$ produce u. Si $m \neq n$, el lado izquierdo aplica `substTerm` a `env m`. Dado que n no ocurre en `env m` (por hipótesis `h_fresh`), `substTerm_fresh` garantiza que la sustitución no tiene efecto, y ambos lados producen `env m`.

- **Casos recursivos:** La propiedad se propaga a los subtérminos mediante las hipótesis de inducción.

Significado

Este lema captura la esencia de la interpretación del cuantificador universal en el modelo canónico. Cuando se evalúa $\forall n\ f$ bajo una asignación σ, se debe verificar f para toda modificación $\sigma[n := t]$. La función `substAll_raw` implementa esta modificación precisamente mediante el entorno modificado $\lambda v.\ \text{if } v = n \text{ then var } n \text{ else env } v$.

El lema muestra que sustituir n por u después de aplicar este entorno modificado es equivalente a aplicar un entorno donde n se mapea directamente a u. Esta equivalencia es la que permite demostrar el caso del cuantificador existencial en el Teorema de Verdad: la existencia de un testigo t tal que $M \models_{\sigma[n:=t]} f$ es equivalente, vía la propiedad de Henkin, a la pertenencia de la fórmula cerrada correspondiente a Δ. La hipótesis `h_fresh` (que n no ocurre en ningún `env v`) se satisface automáticamente cuando `env` asigna términos cerrados, ya que los términos cerrados no contienen variables.

--
==

-- 4. DEFINICIÓN DE satisfies (PARÁMETROS DE MODELO EXPLICITADOS)

```
--
================================================================


def satisfies {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_henkin : HenkinSet Δ) (h_con : Consistent Δ)

  (env : Nat → TermClosed s) : ArithForm s → Prop

 | ArithForm.bottom => False

 | ArithForm.eq t1 t2 =>

  have h1 : isClosed (substAllTerm t1 (fun v => (env v).val)) = true := isClosed_substAllTerm t1 env

  have h2 : isClosed (substAllTerm t2 (fun v => (env v).val)) = true := isClosed_substAllTerm t2 env

  evalTerm Δ h_max h_con (substAllTerm t1 (fun v => (env v).val)) h1 = evalTerm Δ h_max h_con (substAllTerm t2 (fun v => (env v).val)) h2

 | ArithForm.impl f1 f2 =>

  satisfies Δ h_max h_henkin h_con env f1 → satisfies Δ h_max h_henkin h_con env f2

 | ArithForm.neg f =>

  ¬ satisfies Δ h_max h_henkin h_con env f

 | ArithForm.forall_form n f =>

  ∀ t : TermClosed s, satisfies Δ h_max h_henkin h_con (fun v => if v = n then t else env v) f

 | ArithForm.provable_op heq t =>

  have h : isClosed (substAllTerm t (fun v => (env v).val)) = true := isClosed_substAllTerm t env

  provable_op_M h_max h_con heq (evalTerm Δ h_max h_con (substAllTerm t (fun v => (env v).val)) h)

 | ArithForm.true_op heq t =>

  have h : isClosed (substAllTerm t (fun v => (env v).val)) = true := isClosed_substAllTerm t env

  true_op_M h_max h_con heq (evalTerm Δ h_max h_con (substAllTerm t (fun v => (env v).val)) h)

 | ArithForm.list_nth_eq t1 t2 t3 =>
```

```
  have h1 : isClosed (substAllTerm t1 (fun v => (env v).val)) = true := isClosed_substAllTerm t1 env

  have h2 : isClosed (substAllTerm t2 (fun v => (env v).val)) = true := isClosed_substAllTerm t2 env

  have h3 : isClosed (substAllTerm t3 (fun v => (env v).val)) = true := isClosed_substAllTerm t3 env

  list_nth_eq_M h_max h_con

    (evalTerm Δ h_max h_con (substAllTerm t1 (fun v => (env v).val)) h1)

    (evalTerm Δ h_max h_con (substAllTerm t2 (fun v => (env v).val)) h2)

    (evalTerm Δ h_max h_con (substAllTerm t3 (fun v => (env v).val)) h3)


--
============================================================

-- 5. LEMA substAll_forall_form (Y SOPORTE COMPILADO SIN ADVERTENCIAS)

--
============================================================


lemma substTerm_substAllTerm_forall_form_general {s : Nat} (t : ArithTerm s) (n : Nat) (u : ArithTerm s) (env : Nat → ArithTerm s) (h_fresh : ∀ v, NotOccur n (env v)) :

  substTerm n u (substAllTerm t (fun v => if v = n then ArithTerm.var n else env v)) =

  substAllTerm t (fun v => if v = n then u else env v) := by

 induction t with

 | var m =>

  dsimp [substAllTerm]

  by_cases h_mn : m = n

  · subst h_mn

    rw [if_pos rfl, if_pos rfl]

    change (if m = m then u else ArithTerm.var m) = u

    rw [if_pos rfl]

  · rw [if_neg h_mn, if_neg h_mn]
```

```
    rw [substTerm_fresh (env m) n u (h_fresh m)]
  | const c => rfl
  | henkin c => rfl
  | zero => rfl
  | succ t' ih =>
    simp only [substAllTerm, substTerm]
    rw [ih]
  | add t1 t2 ih1 ih2 =>
    simp only [substAllTerm, substTerm]
    rw [ih1, ih2]
  | mul t1 t2 ih1 ih2 =>
    simp only [substAllTerm, substTerm]
    rw [ih1, ih2]
  | exp t1 t2 ih1 ih2 =>
    simp only [substAllTerm, substTerm]
    rw [ih1, ih2]
  | pred t' ih =>
    simp only [substAllTerm, substTerm]
    rw [ih]
```

Lema de conmutación para `substAll_raw` y propiedad de sustitución en `forall_form`

Se completa la infraestructura para el Teorema de Verdad con un lema fundamental de conmutación a nivel de fórmulas y su corolario para la función `substAll` con entornos de términos cerrados.

Lema `substForm_substAll_raw_forall_form_general`: Conmutación general para fórmulas

Extiende el lema `substTerm_substAllTerm_forall_form_general` al nivel de fórmulas. Bajo la misma hipótesis de frescura (que n no ocurre en ninguna imagen de `env`), se tiene:

$$ \text{substForm } n\ u\ (\text{substAll_raw } \varphi\ (\lambda v.\ \text{if } v = n \text{ then var } n \text{ else env } v)) = \text{substAll_raw } \varphi\ (\lambda v.\ \text{if } v = n \text{ then } u \text{ else env } v) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos con términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`):** Se aplica el lema para términos `substTerm_substAllTerm_forall_form_general` a cada subtérmino.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form m f`:** Es el caso más delicado. Se distinguen dos subcasos:

  - **$n = m$:** La sustitución no penetra el cuantificador (pues la variable está ligada). Ambos lados simplifican el entorno de manera compatible.

  - **$n \neq m$:** La sustitución atraviesa el cuantificador. Se reordena el entorno para que la variable m se procese primero, y se aplica la hipótesis de inducción con un entorno modificado `env'` donde m se mapea a $\text{var } m$. La condición de frescura se preserva porque $n \neq m$, por lo que `NotOccur n (var m)`.

Lema `substAll_forall_form`: Caso particular para entornos cerrados

Especializa el lema anterior al caso en que el entorno asigna términos cerrados (como es el caso en la definición de `satisfies`):

$$ \text{substForm } n\ t.\text{val}\ (\text{substAll_raw } \varphi\ (\lambda v.\ \text{if } v = n \text{ then var } n \text{ else } (\text{env } v).\text{val})) = \text{substAll } \varphi\ (\lambda v.\ \text{if } v = n \text{ then } t \text{ else env } v) $$

Estructura de la demostración

1. Se reescribe `substAll` en términos de `substAll_raw` con los valores subyacentes de los términos cerrados.

2. Se aplica `substForm_substAll_raw_forall_form_general`, verificando la condición de frescura: para cada v, n no ocurre en $(\text{env } v).\text{val}$ porque este es un término cerrado (por `NotOccur_of_isClosed`).

Significado para el Teorema de Verdad

Este lema es la pieza técnica central para la demostración del caso del cuantificador existencial en el Teorema de Verdad. Recordemos que $\exists n\ \varphi$ se define como $\neg \forall n\ \neg \varphi$. La satisfacción de $\forall n\ \neg \varphi$ en el modelo bajo asignación σ es:

$$ M \models_\sigma \forall n\ \neg \varphi \iff \forall t \in \text{TermClosed } s,\ M \models_{\sigma[n:=t]} \neg \varphi $$

Utilizando la definición de `satisfies`, el lado derecho se expresa en términos de `substAll` con el entorno modificado. El lema `substAll_forall_form` permite reescribir la instanciación del cuantificador universal con un término cerrado t como una sustitución simple sobre la fórmula `substAll_raw` con el entorno que trata a n como variable libre. Esta reescritura es esencial para conectar la satisfacción en el modelo con la pertenencia a Δ, ya que las fórmulas en Δ son objetos sintácticos (sin asignaciones), y la sustitución simple `substForm` es la operación que permite pasar de una fórmula con variables libres a una fórmula cerrada que puede ser evaluada en Δ.

La condición de frescura (que n no ocurre en los valores del entorno) se satisface automáticamente porque el entorno asigna términos cerrados, y los términos cerrados no contienen variables. Esta es una de las razones por las que la construcción de Henkin utiliza términos cerrados como testigos: garantiza que las operaciones de sustitución son siempre seguras y no requieren verificaciones adicionales de libertad.

```
lemma substForm_substAll_raw_forall_form_general {s : Nat} (φ : ArithForm s) (n : Nat) (u : ArithTerm s) (env : Nat → ArithTerm s) (h_fresh : ∀ v, NotOccur n (env v)) :
  substForm n u (substAll_raw φ (fun v => if v = n then ArithTerm.var n else env v)) =
  substAll_raw φ (fun v => if v = n then u else env v) := by
 induction φ generalizing env with
 | bottom => rfl
 | eq t1 t2 =>
  dsimp [substAll_raw, substForm]
  rw [substTerm_substAllTerm_forall_form_general t1 n u env h_fresh, substTerm_substAllTerm_forall_form_general t2 n u env h_fresh]
 | impl f1 f2 ih1 ih2 =>
  dsimp [substAll_raw, substForm]
  rw [ih1 env h_fresh, ih2 env h_fresh]
 | neg f ih =>
  dsimp [substAll_raw, substForm]
  rw [ih env h_fresh]
 | forall_form m f ih =>
  dsimp [substAll_raw]
  by_cases h_nm : n = m
  · dsimp [substForm]
   rw [if_pos h_nm]
  have h_env : (fun w => if w = m then ArithTerm.var m else if w = n then ArithTerm.var n else env w) =
          (fun w => if w = m then ArithTerm.var m else if w = n then u else env w) := by
    ext w
    by_cases h_wm : w = m
```

```
      · rw [if_pos h_wm, if_pos h_wm]
      · rw [if_neg h_wm, if_neg h_wm]
        have h_wn : ¬ w = n := by
          intro h_eq
          subst h_eq
          exact h_wm h_nm
        rw [if_neg h_wn, if_neg h_wn]
    rw [h_env]
  · dsimp [substForm]
    rw [if_neg h_nm]
    have h_env_swap : (fun w => if w = m then ArithTerm.var m else if w = n then ArithTerm.var n else env w) =
                      (fun w => if w = n then ArithTerm.var n else if w = m then ArithTerm.var m else env w) := by
      ext w
      by_cases h_wn : w = n
      · subst h_wn
        rw [if_neg h_nm, if_pos rfl, if_pos rfl]
      · simp only [if_neg h_wn]
    have h_env_swap_RHS : (fun w => if w = m then ArithTerm.var m else if w = n then u else env w) =
                          (fun w => if w = n then u else if w = m then ArithTerm.var m else env w) := by
      ext w
      by_cases h_wn : w = n
      · subst h_wn
        rw [if_neg h_nm, if_pos rfl, if_pos rfl]
      · simp only [if_neg h_wn]
    rw [h_env_swap, h_env_swap_RHS]
    let env' : Nat → ArithTerm s := fun w => if w = m then ArithTerm.var m else env w
```

```
    have h_fresh' : ∀ w, NotOccur n (env' w) := by
      intro w
      dsimp [env']
      by_cases h_wm : w = m
      · rw [if_pos h_wm]
        exact NotOccur.var m h_nm
      · rw [if_neg h_wm]
        exact h_fresh w
    rw [ih env' h_fresh']
  | provable_op heq t =>
    dsimp [substAll_raw, substForm]
    rw [substTerm_substAllTerm_forall_form_general t n u env h_fresh]
  | true_op heq t =>
    dsimp [substAll_raw, substForm]
    rw [substTerm_substAllTerm_forall_form_general t n u env h_fresh]
  | list_nth_eq t1 t2 t3 =>
    dsimp [substAll_raw, substForm]
    rw [substTerm_substAllTerm_forall_form_general t1 n u env h_fresh,
substTerm_substAllTerm_forall_form_general t2 n u env h_fresh,
substTerm_substAllTerm_forall_form_general t3 n u env h_fresh]


lemma substAll_forall_form {s : Nat} (φ : ArithForm s) (n : Nat) (t : TermClosed s) (env
: Nat → TermClosed s) :
    substForm n t.val (substAll_raw φ (fun v => if v = n then ArithTerm.var n else (env
v).val)) =
    substAll φ (fun v => if v = n then t else env v) := by
  dsimp [substAll]
  have h_eq : (fun v => (if v = n then t else env v).val) = (fun v => if v = n then t.val else
(env v).val) := by
    ext v
```

```
  split_ifs with h_vn
  · subst h_vn; rfl
  · rfl
 rw [h_eq]
 apply substForm_substAll_raw_forall_form_general
 intro v
 apply NotOccur_of_isClosed
 exact (env v).property

end SPA
```

Lemas auxiliares para el Teorema de Verdad: doble negación y cuantificadores

Se demuestran dos resultados deductivos que establecen la equivalencia entre $\forall v\ \Phi$ y $\forall v\ \neg\neg\Phi$ en el sistema SPA. Estos lemas son necesarios para manejar el caso del cuantificador existencial en la demostración del Teorema de Verdad.

Lema `IsFreeFor_var_self`: Una variable es libre para sí misma

Establece formalmente que para cualquier fórmula φ y cualquier variable v, el término $\text{var } v$ es libre para v en φ:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \text{IsFreeFor } v\ (\text{var } v)\ \varphi $$

La demostración es por inducción sobre φ:

- **Casos sin cuantificadores:** `IsFreeFor` es trivialmente verdadero.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Si $v = n$, la sustitución no penetra y la condición es trivial. Si $v \neq n$, se requiere:

  - `NotOccur n (var v)`: cierto porque $n \neq v$.

  - `IsFreeFor v (var v) f`: por hipótesis de inducción.

Intuición

Sustituir una variable por sí misma nunca causa captura de variables, ya que no se introducen variables nuevas. Este lema es un caso particular de `IsFreeFor_closed` (pues $\text{var } v$ no es cerrado), y es necesario para aplicar `forall_elim` instanciando con la misma variable ligada.

Lema `forall_double_neg_elim`: $\forall v\ \neg\neg\Phi \to \forall v\ \Phi$

Establece que en contexto vacío se puede deducir que si para todo v se cumple $\neg\neg\Phi$, entonces para todo v se cumple Φ:

$$ \emptyset \vdash (\forall v\ \neg\neg\Phi) \to (\forall v\ \Phi) $$

La demostración:

1. Se asume $\forall v\ \neg\neg\Phi$ como hipótesis.

2. Se elimina el cuantificador universal instanciando v con $\text{var } v$, lo cual es válido por `IsFreeFor_var_self`.

3. Se obtiene $(\neg\neg\Phi)[v := \text{var } v] = \neg\neg\Phi$ (por `substForm_self_var`).

4. Por `double_neg` se obtiene Φ.

5. Se generaliza sobre v (válido porque el contexto solo contiene la hipótesis, donde v no es libre).

Lema `forall_double_neg_intro`: $\forall v\ \Phi \to \forall v\ \neg\neg\Phi$

Establece la dirección recíproca:

$$ \emptyset \vdash (\forall v\ \Phi) \to (\forall v\ \neg\neg\Phi) $$

La demostración:

1. Se asume $\forall v\ \Phi$ como hipótesis.

2. Se instancia con $\text{var } v$ para obtener Φ.

3. Se asume $\neg\Phi$ y se deriva una contradicción, obteniendo $\neg\neg\Phi$ por `neg_intro`.

4. Se generaliza sobre v.

Significado para el Teorema de Verdad

Estos lemas establecen la equivalencia deductiva entre $\forall v\ \Phi$ y $\forall v\ \neg\neg\Phi$ en el sistema SPA. Esta equivalencia es necesaria para la demostración del caso del cuantificador existencial en el Teorema de Verdad.

Recordemos que $\exists v\ \Phi$ se define como $\neg\forall v\ \neg\Phi$. En el modelo canónico, la satisfacción de $\exists v\ \Phi$ bajo asignación σ significa que existe un término cerrado t tal que $M \models_{\sigma[v:=t]} \Phi$. Para conectar esto con la pertenencia a Δ, se utiliza la propiedad de Henkin, que trata con fórmulas de la forma $\neg\forall v\ \neg\Psi$ (es decir, existenciales). Sin embargo, la propiedad de Henkin se aplica a $\neg\forall v\ \neg\Psi$, no a $\neg\forall v\ \neg\neg\neg\Psi$. Los lemas de doble negación permiten pasar de una formulación a otra, garantizando que la equivalencia lógica se preserva.

Específicamente, cuando se tiene $\neg\forall v\ \neg\Phi \in \Delta$, se puede aplicar la propiedad de Henkin para obtener un testigo t tal que $\neg\Phi[v := t] \in \Delta$, es decir, $\Phi[v := t] \notin \Delta$. Los lemas de doble negación aseguran que esto es equivalente a lo que se necesita para la satisfacción en el modelo.

```
namespace SPA


--
============================================================================
-- SOPORTE AUXILIAR: ISFREEFOR PARA LA VARIABLE AUTO-SUSTITUIDA
--
============================================================================


lemma IsFreeFor_var_self {s : Nat} (φ : ArithForm s) (v : Nat) : IsFreeFor v
(ArithTerm.var v) φ := by
  induction φ generalizing v with
  | bottom => trivial
  | eq t1 t2 => trivial
  | impl f1 f2 ih1 ih2 => exact ⟨ih1 v, ih2 v⟩
  | neg f ih => exact ih v
  | forall_form n f ih =>
    dsimp [IsFreeFor]
    by_cases h : v = n
    · rw [if_pos h]
      exact True.intro
    · rw [if_neg h]
      exact ⟨NotOccur.var v (Ne.symm h), ih v⟩
  | provable_op _ _ => trivial
  | true_op _ _ => trivial
  | list_nth_eq _ _ _ => trivial


--
============================================================================
```

```
-- DERIVACIONES AUXILIARES EN CONTEXTO VACÍO (SIN REQUISITOS DE FRESCURA)
--
============================================================


lemma forall_double_neg_elim {s : Nat} (v : Nat) (Φ : ArithForm s) :
  SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ))) (ArithForm.forall_form v Φ)) := by
 apply SPA_Deduction_Internal.imp_intro
 apply SPA_Deduction_Internal.forall_intro
 · have h_hyp : SPA_Deduction_Internal s (insert (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ))) ∅) (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ))) :=
    SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
  have h_elim := SPA_Deduction_Internal.forall_elim _ v (ArithForm.neg (ArithForm.neg Φ)) (ArithTerm.var v) h_hyp (IsFreeFor_var_self _ v)
  rw [substForm_self_var] at h_elim
  exact SPA_Deduction_Internal.double_neg _ Φ h_elim
 · intro ψ hψ
  rcases hψ with rfl | h_in
  · dsimp [NotFreeForm]
   rw [if_pos rfl]
   exact True.intro
  · nomatch h_in


lemma forall_double_neg_intro {s : Nat} (v : Nat) (Φ : ArithForm s) :
  SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl (ArithForm.forall_form v Φ) (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ)))) := by
 apply SPA_Deduction_Internal.imp_intro
 apply SPA_Deduction_Internal.forall_intro
```

```
  · apply SPA_Deduction_Internal.neg_intro
    have h_hyp_forall : SPA_Deduction_Internal s (insert (ArithForm.neg Φ) (insert (ArithForm.forall_form v Φ) ∅)) (ArithForm.forall_form v Φ) := by
      apply SPA_Deduction_Internal.weaken (insert (ArithForm.forall_form v Φ) ∅) _
      · exact SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
      · exact Set.subset_insert _ _
    have h_elim := SPA_Deduction_Internal.forall_elim _ v Φ (ArithTerm.var v) h_hyp_forall (IsFreeFor_var_self Φ v)
    rw [substForm_self_var] at h_elim
    have h_hyp_neg : SPA_Deduction_Internal s (insert (ArithForm.neg Φ) (insert (ArithForm.forall_form v Φ) ∅)) (ArithForm.neg Φ) :=
      SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
    exact SPA_Deduction_Internal.neg_elim _ Φ h_elim h_hyp_neg
  · intro ψ hψ
    rcases hψ with rfl | h_in
    · dsimp [NotFreeForm]
      rw [if_pos rfl]
      exact True.intro
    · nomatch h_in
```

Lema `neg_forall_iff_neg_forall_double_neg`: Equivalencia de negaciones de cuantificadores bajo maximalidad

Se demuestra un lema técnico fundamental para el Teorema de Verdad: en una teoría maximal consistente Δ, la negación de una fórmula universalmente cuantificada es equivalente a la negación de la misma fórmula con doble negación en el cuerpo.

Enunciado formal

$$ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal } \Delta \land \text{Consistent } \Delta \implies \forall v \in \mathbb{N},\ \forall \Phi \in \text{ArithForm } s, $$

$$ (\neg \forall v\ \Phi) \in \Delta \iff (\neg \forall v\ \neg\neg\Phi) \in \Delta $$

Estructura de la demostración

La prueba establece dos implicaciones, cada una utilizando la maximalidad y la clausura deductiva de Δ:

Dirección $\Rightarrow$: Se asume $\neg \forall v\ \Phi \in \Delta$ y se debe demostrar $\neg \forall v\ \neg\neg\Phi \in \Delta$.

1. Se aplica `neg_intro` con la fórmula $B := \forall v\ \neg\neg\Phi$ como hipótesis a descargar.

2. En el contexto extendido $\Gamma := \Delta \cup \{B\}$:

   - Se tiene B como hipótesis.

- Por el lema `forall_double_neg_elim` (demostrado previamente en contexto vacío y debilitado a Γ), se obtiene $B \to \forall v\ \Phi$.

- Por modus ponens, se deduce $\forall v\ \Phi$.

- Por otro lado, $\neg \forall v\ \Phi$ está disponible en Δ y se debilita a Γ.

- La contradicción entre $\forall v\ \Phi$ y $\neg \forall v\ \Phi$ produce $\bot$.

3. La descarga de B por `neg_intro` da exactamente $\neg B = \neg \forall v\ \neg\neg\Phi$.

Dirección $\Leftarrow$: Simétrica, utilizando `forall_double_neg_intro` en lugar de `forall_double_neg_elim`. Se asume $\neg \forall v\ \neg\neg\Phi \in \Delta$ y se descarga $A := \forall v\ \Phi$ para obtener $\neg \forall v\ \Phi$.

Significado para el Teorema de Verdad

Este lema es una pieza indispensable en la demostración del caso del cuantificador existencial en el Teorema de Verdad. La razón es la siguiente:

En el modelo canónico, la satisfacción de una fórmula existencial $\exists v\ \Phi$ (que se define como $\neg \forall v\ \neg \Phi$) se traduce, vía la hipótesis de inducción, en la pertenencia a Δ de una fórmula de la forma $\neg \forall v\ \neg \Psi$. Para aplicar la propiedad de Henkin (que proporciona testigos para fórmulas existenciales), se necesita que la fórmula tenga exactamente la forma $\neg \forall v\ \neg (\text{algo})$, es decir, con una negación inmediatamente después del cuantificador universal.

Sin embargo, durante la demostración del caso `forall_form` en el Teorema de Verdad, la hipótesis de inducción produce una fórmula $\neg \forall v\ \Psi$ donde Ψ puede no tener la forma $\neg \Psi'$. El lema `neg_forall_iff_neg_forall_double_neg` permite reemplazar $\neg \forall v\ \Psi$ por $\neg \forall v\ \neg\neg\Psi$, que sí tiene la forma requerida para aplicar la propiedad de Henkin (con $\neg\Psi$ como la fórmula interna). Esto cierra la brecha entre lo que proporciona la hipótesis de inducción y lo que necesita la propiedad de Henkin.

La demostración del lema utiliza de manera esencial la maximalidad de Δ (para garantizar que las fórmulas pertenecen a Δ por clausura deductiva) y los lemas `forall_double_neg_elim` e `forall_double_neg_intro` (que establecen la equivalencia deductiva en contexto vacío). La combinación de estos ingredientes permite transferir la equivalencia sintáctica a una equivalencia de pertenencia en Δ.

```
--
================================================================

-- LEMA DE EQUIVALENCIA PARA LA MAXIMALIDAD

--
================================================================


lemma neg_forall_iff_neg_forall_double_neg {s : Nat} {Δ : Set (ArithForm s)} (h_max :
Maximal Δ) (h_con : Consistent Δ) (v : Nat) (Φ : ArithForm s) :

  (ArithForm.neg (ArithForm.forall_form v Φ)) ∈ Δ ↔ (ArithForm.neg
(ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ)))) ∈ Δ := by

 constructor

 · intro h_in
```

```
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.neg_intro
  let B := ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ))
  let Γ := insert B Δ
  have h1 : SPA_Deduction_Internal s Γ (ArithForm.impl (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ))) (ArithForm.forall_form v Φ)) := by
    apply SPA_Deduction_Internal.weaken (∅ : Set (ArithForm s)) _ Γ (forall_double_neg_elim v Φ) (Set.empty_subset _)
  have h2 : SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ))) :=
    SPA_Deduction_Internal.hyp Γ _ (Or.inl rfl)
  have h_forall_Phi := SPA_Deduction_Internal.imp_elim Γ _ _ h1 h2
  have h_neg_forall : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.forall_form v Φ)) := by
    apply SPA_Deduction_Internal.weaken Δ _ Γ (SPA_Deduction_Internal.hyp Δ _ h_in) (Set.subset_insert _ _)
  exact SPA_Deduction_Internal.neg_elim Γ (ArithForm.forall_form v Φ) h_forall_Phi h_neg_forall
· intro h_in
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.neg_intro
  let A := ArithForm.forall_form v Φ
  let Γ := insert A Δ
  have h1 : SPA_Deduction_Internal s Γ (ArithForm.impl (ArithForm.forall_form v Φ) (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ)))) := by
    apply SPA_Deduction_Internal.weaken (∅ : Set (ArithForm s)) _ Γ (forall_double_neg_intro v Φ) (Set.empty_subset _)
  have h2 : SPA_Deduction_Internal s Γ (ArithForm.forall_form v Φ) :=
    SPA_Deduction_Internal.hyp Γ _ (Or.inl rfl)
  have h_forall_nnPhi := SPA_Deduction_Internal.imp_elim Γ _ _ h1 h2
  have h_neg_forall : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ)))) := by
```

apply SPA_Deduction_Internal.weaken Δ _ Γ (SPA_Deduction_Internal.hyp Δ _ h_in) (Set.subset_insert _ _)

exact SPA_Deduction_Internal.neg_elim Γ (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Φ))) h_forall_nnPhi h_neg_forall

El Teorema de Verdad (Truth Lemma)

Se presenta la demostración completa del Teorema de Verdad para el modelo canónico, que constituye el núcleo de la prueba de completitud. Este teorema establece que la verdad en el modelo canónico coincide exactamente con la pertenencia a la teoría maximal consistente Δ.

Enunciado formal

Para toda teoría Δ maximal consistente con propiedad de Henkin, toda asignación $\sigma : \mathbb{N} \to \text{TermClosed } s$, y toda fórmula φ:

$$ M(\Delta) \models_\sigma \varphi \iff \text{substAll } \varphi\ \sigma \in \Delta $$

Demostración por inducción estructural

Caso `bottom`: $\bot$ nunca se satisface en el modelo (por definición). Si $\bot \in \Delta$, entonces Δ sería inconsistente (por `h_con`), contradiciendo la hipótesis de consistencia.

Caso `eq t1 t2`: La satisfacción de $t_1 = t_2$ en el modelo se define como la igualdad de las evaluaciones de t_1 y t_2 en $M(\Delta)$. Por definición de `evalTerm` y la construcción del cociente, esto equivale exactamente a $(t_1 = t_2) \in \Delta$. La dirección $\Rightarrow$ usa `Quotient.exact` (si dos clases son iguales, los representantes son equivalentes); la dirección $\Leftarrow$ usa `Quotient.sound` (si dos representantes son equivalentes, sus clases son iguales).

Caso `impl f1 f2`:

- **Dirección $\Rightarrow$:** Se asume $M \models f_1 \to f_2$ y se supone, por contradicción, que $(f_1 \to f_2) \notin \Delta$. Por maximalidad, $\neg(f_1 \to f_2) \in \Delta$. Utilizando los lemas `imp_neg_elim_left` e `imp_neg_elim_right`, se deducen $f_1 \in \Delta$ y $\neg f_2 \in \Delta$. Por hipótesis de inducción, $M \models f_1$ y $M \not\models f_2$, lo que contradice $M \models f_1 \to f_2$.

- **Dirección $\Leftarrow$:** Si $f_1 \to f_2 \in \Delta$ y $M \models f_1$, por hipótesis de inducción $f_1 \in \Delta$. Por `maximal_consistent_imp_closed`, $f_2 \in \Delta$, y por hipótesis de inducción $M \models f_2$.

Caso `neg f`:

- **Dirección $\Rightarrow$:** Si $M \not\models f$, entonces $f \notin \Delta$ (por hipótesis de inducción). Por maximalidad, $\neg f \in \Delta$.

- **Dirección $\Leftarrow$:** Si $\neg f \in \Delta$ pero $M \models f$, entonces $f \in \Delta$ (por hipótesis de inducción), y la presencia simultánea de f y $\neg f$ en Δ contradice la consistencia.

Caso `forall_form n f`:

Sea $\Psi := \text{substAll_raw } f\ (\lambda v.\ \text{if } v = n \text{ then var } n \text{ else } (\sigma\ v).\text{val})$.

- **Dirección $\Rightarrow$:** Se asume que para todo t, $M \models_{\sigma[n:=t]} f$. Si $\forall n\ \Psi \notin \Delta$, por maximalidad $\neg\forall n\ \Psi \in \Delta$. Por el lema `neg_forall_iff_neg_forall_double_neg`, $\neg\forall n\ \neg\neg\Psi \in \Delta$. Aplicando la propiedad de Henkin a esta fórmula existencial, existe un testigo t tal que $\neg\Psi[n := t] \in \Delta$. Pero por el lema `substAll_forall_form`, $\Psi[n := t] = \text{substAll } f\ (\sigma[n:=t])$. Por hipótesis de inducción, $M \models_{\sigma[n:=t]} f$ implica $\Psi[n := t] \in \Delta$. La presencia simultánea de $\Psi[n := t]$ y $\neg\Psi[n := t]$ en Δ contradice la consistencia.

- **Dirección $\Leftarrow$:** Si $\forall n\ \Psi \in \Delta$, para cualquier t se aplica `forall_elim` (usando `IsFreeFor_closed`) para obtener $\Psi[n := t] \in \Delta$. Por `substAll_forall_form` e hipótesis de inducción, $M \models_{\sigma[n:=t]} f$.

Casos `provable_op`, `true_op`, `list_nth_eq`: Se siguen directamente de las definiciones, ya que las interpretaciones en el modelo se definieron precisamente como la pertenencia a Δ de las fórmulas correspondientes.

Significado

El Teorema de Verdad es el resultado culminante de la construcción del modelo canónico. Establece una correspondencia perfecta entre la semántica (satisfacción en el modelo) y la sintaxis (pertenencia a la teoría maximal). Esta correspondencia es la que permite completar la prueba de completitud:

1. Se parte de una teoría consistente Γ.

2. Se construye su extensión de Henkin Γ_ω (mediante `GammaLimit`).

3. Se extiende a una teoría maximal consistente Δ (mediante `LindenbaumLimit`).

4. Se construye el modelo canónico $M(\Delta)$ (mediante `model_from_maximal`).

5. Por el Teorema de Verdad, $M(\Delta) \models \varphi \iff \varphi \in \Delta$.

6. Como $\Gamma \subseteq \Delta$ (por `LindenbaumLimit_contains_Gamma`), se tiene $M(\Delta) \models \Gamma$.

Por tanto, Γ es satisfacible, lo que demuestra que toda teoría consistente tiene un modelo. Este es el Teorema de Completitud de Gödel para el sistema estratificado SPA.

La elegancia de la demostración radica en la construcción puramente sintáctica del modelo: el dominio son los términos cerrados del lenguaje, las operaciones se definen sintácticamente, y la verdad se reduce a la pertenencia a la teoría. No se requiere apelar a nociones externas de verdad o infinitud; el modelo emerge de la sintaxis misma. Este es el legado del método de Henkin-Lindenbaum.

```
--
================================================================

-- EL TEOREMA DE VERDAD (TRUTH LEMMA) COMPLETADO

--
================================================================


theorem truth_lemma {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_henkin :
HenkinSet Δ) (h_con : Consistent Δ)

   (env : Nat → TermClosed s) (φ : ArithForm s) :

   satisfies Δ h_max h_henkin h_con env φ ↔ substAll φ env ∈ Δ := by

 induction φ generalizing env with

 | bottom =>

   dsimp [satisfies, substAll, substAll_raw]

   constructor

   · intro h; contradiction

   · intro h

    have h_deriv : SPA_Deduction_Internal s Δ ArithForm.bottom :=
SPA_Deduction_Internal.hyp Δ ArithForm.bottom h

     exact h_con h_deriv

 | eq t1 t2 =>

   dsimp [satisfies, substAll, substAll_raw, evalTerm]

   constructor

   · intro h

     exact Quotient.exact h

   · intro h
```

```
    exact Quotient.sound h
  | impl f1 f2 ih1 ih2 =>
    dsimp [satisfies, substAll, substAll_raw]
    constructor
    · intro h_cond
      by_contra h_not_in
      have h_neg_in : ArithForm.neg (ArithForm.impl (substAll f1 env) (substAll f2 env)) ∈ Δ := by
        cases h_max (ArithForm.impl (substAll f1 env) (substAll f2 env)) with
        | inl h => contradiction
        | inr h => exact h
      have h_neg_ded : SPA_Deduction_Internal s Δ (ArithForm.neg (ArithForm.impl (substAll f1 env) (substAll f2 env))) :=
        SPA_Deduction_Internal.hyp Δ _ h_neg_in
      have h_A_ded := imp_neg_elim_left Δ _ _ h_neg_ded
      have h_A : substAll f1 env ∈ Δ := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_A_ded
      have h_B : substAll f2 env ∈ Δ := Iff.mp (ih2 env) (h_cond (Iff.mpr (ih1 env) h_A))
      have h_neg_B_ded := imp_neg_elim_right Δ _ _ h_neg_ded
      have h_neg_B : ArithForm.neg (substAll f2 env) ∈ Δ := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_neg_B_ded
      have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
        SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_B) (SPA_Deduction_Internal.hyp Δ _ h_neg_B)
      exact h_con h_bot
    · intro h_imp h_f1
      have h_A : substAll f1 env ∈ Δ := Iff.mp (ih1 env) h_f1
      exact Iff.mpr (ih2 env) (maximal_consistent_imp_closed Δ h_max h_con _ _ h_A h_imp)
  | neg f ih =>
```

```
  dsimp [satisfies, substAll, substAll_raw]
  constructor
  · intro h
    have h_not : substAll f env ∉ Δ := by
      intro h_in
      exact h (Iff.mpr (ih env) h_in)
    cases h_max (substAll f env) with
    | inl h_in => contradiction
    | inr h_neg_in => exact h_neg_in
  · intro h h_sat
    have h_in : substAll f env ∈ Δ := Iff.mp (ih env) h_sat
    have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
      SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_in) (SPA_Deduction_Internal.hyp Δ _ h)
    exact h_con h_bot
| forall_form n f ih =>
  let Ψ := substAll_raw f (fun v => if v = n then ArithTerm.var n else (env v).val)
  change (∀ t : TermClosed s, satisfies Δ h_max h_henkin h_con (fun v => if v = n then t else env v) f) ↔
      ArithForm.forall_form n Ψ ∈ Δ
  constructor
  · intro h
    by_contra h_not_in
    have h_neg_in : ArithForm.neg (ArithForm.forall_form n Ψ) ∈ Δ := by
      cases h_max (ArithForm.forall_form n Ψ) with
      | inl h => contradiction
      | inr h => exact h
  have h_neg_double : ArithForm.neg (ArithForm.forall_form n (ArithForm.neg (ArithForm.neg Ψ))) ∈ Δ := by
```

```
      rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con n Ψ]
    obtain ⟨t, h_witness⟩ := h_henkin n (ArithForm.neg Ψ) h_neg_double
    have h_neg_subst : ArithForm.neg (substForm n t.val Ψ) ∈ Δ := h_witness
    have h_sat := h t
    have h_subst_in := Iff.mp (ih (fun v => if v = n then t else env v)) h_sat
    rw [← substAll_forall_form] at h_subst_in
    have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
      SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_subst_in)
(SPA_Deduction_Internal.hyp Δ _ h_neg_subst)
    exact h_con h_bot
  · intro h t
    have h_forall : SPA_Deduction_Internal s Δ (ArithForm.forall_form n Ψ) :=
SPA_Deduction_Internal.hyp Δ _ h
    have h_elim := SPA_Deduction_Internal.forall_elim Δ n Ψ t.val h_forall
(IsFreeFor_closed Ψ n t)
    rw [substAll_forall_form] at h_elim
    exact Iff.mpr (ih (fun v => if v = n then t else env v))
(maximal_consistent_closed_under_deduction Δ h_max h_con _ h_elim)
 | provable_op heq t =>
   dsimp [satisfies, substAll, substAll_raw]
   rfl
 | true_op heq t =>
   dsimp [satisfies, substAll, substAll_raw]
   rfl
 | list_nth_eq t1 t2 t3 =>
   dsimp [satisfies, substAll, substAll_raw]
   rfl
```

Lemas de conmutación de sustituciones y preservación de libertad

Se demuestran dos lemas técnicos que establecen propiedades fundamentales sobre la interacción entre sustituciones y la condición de libertad (`IsFreeFor`). Estos resultados son herramientas esenciales para la manipulación de fórmulas con sustituciones múltiples en el sistema deductivo.

Lema `IsFreeFor_subst_term`: Preservación de libertad bajo sustitución

Establece que si un término u es libre para una variable w en una fórmula φ, entonces al aplicar una sustitución sobre una variable distinta v con un término cerrado t, el término resultante sigue siendo libre para w en φ:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \forall t \in \text{TermClosed } s,\ \forall u \in \text{ArithTerm } s, $$

$$ \text{IsFreeFor } w\ u\ \varphi \implies \text{IsFreeFor } w\ (\text{substTerm } v\ t.\text{val}\ u)\ \varphi $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos sin cuantificadores:** Trivial, pues `IsFreeFor` no depende del término u para estos constructores.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción a las subfórmulas.

- **`forall_form n f`:** Si $w = n$, la sustitución no penetra y la condición es trivial. Si $w \neq n$, la hipótesis `IsFreeFor w u (forall_form n f)` proporciona `NotOccur n u` (la variable ligada no ocurre en u) e `IsFreeFor w u f`. Al sustituir v por $t.\text{val}$ en u, se debe verificar que n sigue sin ocurrir en el resultado. Esto se sigue de `NotOccur_subst_other`: como n no ocurre en u ni en $t.\text{val}$ (este último por ser cerrado, `NotOccur_of_isClosed`), n no ocurre en $\text{substTerm } v\ t.\text{val}\ u$. La hipótesis de inducción proporciona la preservación de la libertad en f.

Significado

Este lema garantiza que la propiedad de "ser libre para" es estable bajo sustituciones que involucran términos cerrados sobre variables distintas. Es utilizado en contextos donde se necesita aplicar múltiples sustituciones sucesivas y se debe verificar que la condición de libertad para la aplicación de `forall_elim` se mantiene.

Lema `substTerm_comm`: Conmutación de sustituciones independientes

Establece que dos sustituciones sobre variables distintas conmutan, bajo la condición adicional de que el término de la primera sustitución no contiene la variable de la segunda:

$$ \forall t' \in \text{ArithTerm } s,\ \forall v, x \in \mathbb{N},\ v \neq x \land \text{NotOccur } x\ t \implies $$

$$ \text{substTerm } v\ t\ (\text{substTerm } x\ u\ t') = \text{substTerm } x\ (\text{substTerm } v\ t\ u)\ (\text{substTerm } v\ t\ t') $$

Estructura de la demostración

Por inducción sobre t':

- **`var m`:** Se analiza mediante casos sobre si $m = x$ y si $m = v$:

  - Si $m = x$: Como $v \neq x$, $m \neq v$. La sustitución interna reemplaza $\text{var } x$ por u; la externa aplica $\text{substTerm } v\ t$ a u y deja el resto igual. El lado derecho produce el mismo resultado.

  - Si $m = v$: La sustitución interna no afecta (pues $m \neq x$), la externa reemplaza $\text{var } v$ por t. El lado derecho produce t en ambos lugares, y como x no ocurre en t (por `h_not_occur`), `substTerm x ... t = t`.

  - Si m es distinto de ambos: Ninguna sustitución afecta.

- **Casos recursivos:** Se aplican las hipótesis de inducción a los subtérminos.

Significado

Este lema es fundamental para la manipulación de términos con múltiples variables libres. Establece que el orden de sustitución de variables distintas es irrelevante (módulo la sustitución interna del término de reemplazo), siempre que no haya interferencia entre ellas. La condición `NotOccur x t` es crucial: garantiza que la primera sustitución no introduce la variable x, lo que podría causar una captura indeseada al aplicar la segunda sustitución. Esta condición se satisface automáticamente cuando t es un término cerrado, ya que los términos cerrados no contienen variables. Ambos lemas son utilizados en las demostraciones de representabilidad y en la manipulación de fórmulas dentro del sistema deductivo.

```
--
================================================================
-- LEMAS DE CONMUTACIÓN DE SUSTITUCIONES (TÉRMINOS Y FÓRMULAS)
--
================================================================


--
================================================================
-- 1. LEMA DE SOPORTE PARA IsFreeFor (Preservación de sustitución de términos)
--
================================================================


lemma IsFreeFor_subst_term {s : Nat} (φ : ArithForm s) (v w : Nat) (t : TermClosed s)
(u : ArithTerm s)
    (h : IsFreeFor w u φ) : IsFreeFor w (substTerm v t.val u) φ := by
  unfold TermClosed at t
  induction φ generalizing v w u with
  | bottom => trivial
  | eq _ _ => trivial
  | impl f1 f2 ih1 ih2 =>
    dsimp [IsFreeFor] at h ⊢
```

```
  exact ⟨ih1 v w u h.1, ih2 v w u h.2⟩
 | neg f ih =>
  dsimp [IsFreeFor] at h ⊢
  exact ih v w u h
 | forall_form n f ih =>
  dsimp [IsFreeFor] at h ⊢
  by_cases h_wn : w = n
  · rw [if_pos h_wn]
    trivial
  · rw [if_neg h_wn] at h ⊢
    exact ⟨NotOccur_subst_other u v n t.val h.1 (NotOccur_of_isClosed t.val t.property
n), ih v w u h.2⟩
 | provable_op h t' => trivial
 | true_op h t' => trivial
 | list_nth_eq t1 t2 t3 => trivial


--
============================================================================
-- 2. CORRECCIÓN EN substTerm_comm (Uso de by_cases determinista)
--
============================================================================


lemma substTerm_comm {s : Nat} (t' : ArithTerm s) (v x : Nat) (t u : ArithTerm s)
  (h_ne : v ≠ x) (h_not_occur : NotOccur x t) :
  substTerm v t (substTerm x u t') = substTerm x (substTerm v t u) (substTerm v t t')
:= by
 induction t' with
 | var m =>
  by_cases h1 : x = m
  · by_cases h2 : v = m
```

```
    · subst h1 h2
      exact (h_ne rfl).elim
    · simp only [substTerm_var, if_pos h1, if_neg h2]
  · by_cases h2 : v = m
    · simp only [substTerm_var, if_neg h1, if_pos h2, substTerm_fresh t x _ h_not_occur]
    · simp only [substTerm_var, if_neg h1, if_neg h2]
| const c => try rfl
| henkin c => try rfl
| zero => try rfl
| succ t'' ih => dsimp [substTerm]; rw [ih]
| add t1 t2 ih1 ih2 => dsimp [substTerm]; rw [ih1, ih2]
| mul t1 t2 ih1 ih2 => dsimp [substTerm]; rw [ih1, ih2]
| exp t1 t2 ih1 ih2 => dsimp [substTerm]; rw [ih1, ih2]
| pred t'' ih => dsimp [substTerm]; rw [ih]
```

Lemas de conmutación y asociatividad de sustituciones

Se demuestran tres lemas fundamentales que completan el álgebra de sustituciones del sistema SPA: la conmutación de sustituciones independientes a nivel de fórmulas, y la propiedad asociativa de las sustituciones sobre una misma variable.

Lema `substForm_comm`: Conmutación de sustituciones independientes en fórmulas

Extiende `substTerm_comm` al nivel de fórmulas: dos sustituciones sobre variables distintas conmutan, bajo condiciones de no interferencia:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, x \in \mathbb{N},\ v \neq x \land \text{NotOccur } x\ t \land \text{NotOccur } v\ u \implies $$

$$ \text{substForm } v\ t\ (\text{substForm } x\ u\ \varphi) = \text{substForm } x\ (\text{substTerm } v\ t\ u)\ (\text{substForm } v\ t\ \varphi) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos con términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`):** Se aplica `substTerm_comm` a cada subtérmino.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Se analiza si $x = n$ o $v = n$:

  - Si $x = n$: La sustitución sobre x no penetra el cuantificador. Como $v \neq x$, $v \neq n$, y la sustitución sobre v atraviesa el cuantificador. El resultado es el mismo en ambos lados.

  - Si $v = n$: Análogo, pero la sustitución sobre v no penetra. Se utiliza además `substTerm_fresh` porque v no ocurre en u (por `h_not_occur_vu`), por lo que $\text{substTerm } v\ t\ u = u$.

  - Si ambos son distintos de n: Ambas sustituciones atraviesan el cuantificador, y la hipótesis de inducción se aplica al cuerpo f.

Significado

Este lema es la versión para fórmulas de la propiedad de que las sustituciones independientes conmutan. Las condiciones de no ocurrencia garantizan que no hay interferencia entre las dos operaciones: la primera sustitución no introduce la variable de la segunda, y viceversa. Esto es siempre cierto cuando los términos de sustitución son cerrados (no contienen variables), lo cual es el caso más común en las demostraciones de representabilidad.

Lema `substTerm_substTerm_same`: Asociatividad de sustituciones sobre la misma variable

Establece que sustituir n por u y luego n por t (donde t puede contener a n) es equivalente a sustituir n por t directamente, aplicando primero la sustitución interna a t:

$$ \forall t' \in \text{ArithTerm } s,\ \forall n \in \mathbb{N},\ \forall t, u \in \text{ArithTerm } s, $$

$$ \text{substTerm } n\ (\text{substTerm } n\ t\ u)\ t' = \text{substTerm } n\ t\ (\text{substTerm } n\ u\ t') $$

La demostración es por inducción sobre t', con un análisis de casos para `var m` según si $m = n$.

Lema `substForm_substForm_same`: Asociatividad para fórmulas

Extiende el lema anterior al nivel de fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall n \in \mathbb{N},\ \forall t, u \in \text{ArithTerm } s, $$

$$ \text{substForm } n\ (\text{substTerm } n\ t\ u)\ \varphi = \text{substForm } n\ t\ (\text{substForm } n\ u\ \varphi) $$

La demostración es por inducción sobre φ, utilizando `substTerm_substTerm_same` para los casos con términos. En el caso `forall_form m f`, si $n = m$ la sustitución no penetra (ambos lados preservan la fórmula); si $n \neq m$, ambas sustituciones atraviesan el cuantificador y se aplica la hipótesis de inducción.

Significado conjunto

Estos tres lemas completan el álgebra de sustituciones del sistema SPA, estableciendo propiedades análogas a las de un sistema de reescritura: conmutatividad de operaciones independientes y asociatividad de operaciones sobre la misma variable. Estas propiedades son fundamentales para la manipulación de fórmulas con múltiples cuantificadores y sustituciones, como las que aparecen en `NumForm`, `BetaForm`, y las demostraciones de representabilidad. En particular, `substForm_comm` es utilizado en `subst_comm_closed` y en las demostraciones de conmutación de sustituciones para la composición de fórmulas representables, mientras que `substForm_substForm_same` se utiliza en las simplificaciones de sustituciones encadenadas sobre la misma variable.

```
--
================================================================
-- 3. CORRECCIÓN EN substForm_comm (Uso de exact rfl para bottom)
--
================================================================

lemma substForm_comm {s : Nat} (φ : ArithForm s) (v x : Nat) (t u : ArithTerm s)
   (h_ne : v ≠ x) (h_not_occur : NotOccur x t) (h_not_occur_vu : NotOccur v u) :
   substForm v t (substForm x u φ) = substForm x (substTerm v t u) (substForm v t φ)
:= by
  induction φ generalizing u with
  | bottom => any_goals rfl
  | eq t1 t2 =>
   rw [substForm_eq, substForm_eq, substForm_eq, substForm_eq]
   rw [substTerm_comm t1 v x t u h_ne h_not_occur, substTerm_comm t2 v x t u h_ne
h_not_occur]
  | impl f1 f2 ih1 ih2 =>
   simp only [substForm_impl, ih1 u h_not_occur_vu, ih2 u h_not_occur_vu]
  | neg f ih =>
   simp only [substForm_neg, ih u h_not_occur_vu]
  | forall_form n f ih =>
   rw [substForm_forall_form]
   by_cases h_xn : x = n
   · have h_vn : v ≠ n := by
      intro h_eq
      exact h_ne (h_eq.trans h_xn.symm)
    simp only [if_pos h_xn, if_neg h_vn, substForm_forall_form]
   · by_cases h_vn : v = n
    · simp only [if_neg h_xn, if_pos h_vn, substForm_forall_form]
      rw [substTerm_fresh u v t h_not_occur_vu]
```

```
    · simp only [if_neg h_xn, if_neg h_vn, substForm_forall_form]
      rw [ih u h_not_occur_vu]
  | provable_op h t' =>
    simp only [substForm_provable_op, substTerm_comm t' v x t u h_ne h_not_occur]
  | true_op h t' =>
    simp only [substForm_true_op, substTerm_comm t' v x t u h_ne h_not_occur]
  | list_nth_eq t1 t2 t3 =>
    simp only [substForm_list_nth_eq, substTerm_comm t1 v x t u h_ne h_not_occur,
substTerm_comm t2 v x t u h_ne h_not_occur, substTerm_comm t3 v x t u h_ne
h_not_occur]



--
============================================================================
-- 3. REDEFINICIÓN ROBUSTA DE substTerm_substTerm_same (Evita rws
redundantes)
--
============================================================================


lemma substTerm_substTerm_same {s : Nat} (t' : ArithTerm s) (n : Nat) (t u :
ArithTerm s) :
    substTerm n (substTerm n t u) t' = substTerm n t (substTerm n u t') := by
  induction t' with
  | var m =>
    by_cases h_nm : n = m
    · subst h_nm
      simp [substTerm_var]
    · simp [substTerm_var, h_nm]
  | const c => try rfl
  | henkin c => try rfl
```

```
  | zero => try rfl
  | succ t'' ih => dsimp [substTerm]; rw [ih]
  | add t1 t2 ih1 ih2 => dsimp [substTerm]; rw [ih1, ih2]
  | mul t1 t2 ih1 ih2 => dsimp [substTerm]; rw [ih1, ih2]
  | exp t1 t2 ih1 ih2 => dsimp [substTerm]; rw [ih1, ih2]
  | pred t'' ih => dsimp [substTerm]; rw [ih]


lemma substForm_substForm_same {s : Nat} (φ : ArithForm s) (n : Nat) (t u :
ArithTerm s) :
    substForm n (substTerm n t u) φ = substForm n t (substForm n u φ) := by
  induction φ generalizing u with
  | bottom => rfl
  | eq t1 t2 =>
    dsimp [substForm]
    rw [substTerm_substTerm_same t1 n t u, substTerm_substTerm_same t2 n t u]
  | impl f1 f2 ih1 ih2 =>
    dsimp [substForm]
    rw [ih1 u, ih2 u]
  | neg f ih =>
    dsimp [substForm]
    rw [ih u]
  | forall_form m f ih =>
    by_cases h_nm : n = m
    · subst h_nm
      rw [substForm]
      rw [if_pos rfl]
      rw [substForm]
      rw [if_pos rfl]
```

```
    rw [substForm]
    rw [if_pos rfl]
  · rw [substForm, substForm]
    rw [if_neg h_nm, if_neg h_nm]
    rw [substForm]
    rw [if_neg h_nm]
    rw [ih u]
| provable_op heq t' =>
  dsimp [substForm]
  rw [substTerm_substTerm_same t' n t u]
| true_op heq t' =>
  dsimp [substForm]
  rw [substTerm_substTerm_same t' n t u]
| list_nth_eq t1 t2 t3 =>
  dsimp [substForm]
  rw [substTerm_substTerm_same t1 n t u, substTerm_substTerm_same t2 n t u,
substTerm_substTerm_same t3 n t u]
```

Lemas de frescura bajo sustitución y preservación en conjuntos

Se demuestran tres lemas que establecen cómo las propiedades de frescura (no ocurrencia y no libertad) se preservan bajo operaciones de sustitución, y cómo estas propiedades se transfieren a conjuntos de fórmulas transformados.

Lema `NotOccur_substTerm`: Preservación de no ocurrencia en términos bajo sustitución

Establece que si una variable w no ocurre en un término t', entonces w tampoco ocurre en t' tras sustituir cualquier variable v por un término cerrado t:

$$ \forall t' \in \text{ArithTerm } s,\ \forall v, w \in \mathbb{N},\ \forall t \in \text{ArithTerm } s, $$

$$ \text{NotOccur } w\ t' \land \text{isClosed } t = \text{true} \implies \text{NotOccur } w\ (\text{substTerm } v\ t\ t') $$

Estructura de la demostración

Por inducción sobre t':

- **`var n`:** Si $v = n$, la variable se reemplaza por t. Como t es cerrado, `NotOccur_of_isClosed` garantiza que w no ocurre en t. Si $v \neq n$, la variable se preserva y $w \neq n$ por la hipótesis `NotOccur w (var n)`.

- **Casos base (`const`, `henkin`, `zero`):** Triviales.

- **Casos recursivos:** Se invierte la hipótesis `NotOccur w t'` y se aplican las hipótesis de inducción a los subtérminos.

Significado

Este lema es fundamental para garantizar que las operaciones de sustitución no introducen variables no deseadas. Junto con `NotOccur_subst_other` y `NotOccur_subst_self`, forma una familia de resultados que caracterizan completamente el comportamiento de `NotOccur` bajo sustituciones.

Lema `NotFreeForm_substForm`: Preservación de no libertad en fórmulas bajo sustitución

Establece que si una variable w no es libre en una fórmula φ, entonces w sigue sin ser libre en φ tras sustituir cualquier variable v por un término cerrado t:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \forall t \in \text{ArithTerm } s, $$

$$ \text{NotFreeForm } w\ \varphi \land \text{isClosed } t = \text{true} \implies \text{NotFreeForm } w\ (\text{substForm } v\ t\ \varphi) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos con términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`):** Se aplica `NotOccur_subst_other` a cada subtérmino, usando que t es cerrado.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Se analiza si $v = n$:

- Si $v = n$: La sustitución no penetra, y la fórmula no cambia. La no libertad de w se preserva directamente de la hipótesis. Si $w = n$, w está ligada y por tanto no es libre; si $w \neq n$, w no era libre en f y sigue sin serlo.

- Si $v \neq n$: La sustitución atraviesa el cuantificador. Si $w = n$, w está ligada y no es libre (trivial). Si $w \neq n$, w no era libre en f, y por hipótesis de inducción sigue sin serlo en $\text{substForm } v\ t\ f$.

Significado

Este lema es crucial para la regla de generalización universal (`forall_intro`). Cuando se aplica `forall_intro` para obtener $\forall v\ \varphi$, se requiere que v no sea libre en las hipótesis. Si posteriormente se realiza una sustitución sobre otra variable, este lema garantiza que la condición de no libertad se preserva, permitiendo que la generalización siga siendo válida en el contexto transformado.

Lema `NotFreeInSet_substForm`: Preservación en conjuntos

Extiende la propiedad anterior a conjuntos de fórmulas: si una variable w no es libre en ninguna fórmula de Γ, entonces w tampoco es libre en las fórmulas resultantes de aplicar una sustitución (con término cerrado) a Γ:

$$ \forall v, w \in \mathbb{N},\ \forall t \in \text{TermClosed } s,\ \forall \Gamma \subseteq \text{ArithForm } s, $$

$$ \text{NotFreeInSet } w\ \Gamma \implies \text{NotFreeInSet } w\ \{\text{substForm } v\ t.\text{val}\ \psi \mid \psi \in \Gamma\} $$

La demostración es directa: para cada fórmula ψ' en el conjunto imagen, existe $\psi \in \Gamma$ tal que $\psi' = \text{substForm } v\ t.\text{val}\ \psi$, y se aplica `NotFreeForm_substForm` usando que w no es libre en ψ y que t es cerrado.

Significado conjunto

Estos lemas forman la base de la maquinaria de "preservación de frescura bajo transformaciones sintácticas" que es ubicua en la prueba de completitud. Garantizan que las propiedades de frescura necesarias para las reglas de inferencia (como `forall_intro`, `forall_elim`, y los lemas de cambio de variable) se mantienen cuando las fórmulas son modificadas mediante sustituciones, lo cual es indispensable para la eliminación de constantes de Henkin y la construcción del modelo canónico. La condición de que el término de sustitución sea cerrado es recurrente y refleja el hecho de que las constantes de Henkin y los numerales no introducen variables libres, haciendo que las sustituciones sean "inocuas" respecto a la frescura.

```
--
================================================================

-- PROPIEDADES DE FRESCURA Y OPERACIÓN SOBRE CONJUNTOS

--
================================================================


lemma NotOccur_substTerm {s : Nat} (t' : ArithTerm s) (v : Nat) (w : Nat) (t :
ArithTerm s)

   (h_occ : NotOccur w t') (h_closed : isClosed t = true) :
```

```
  NotOccur w (substTerm v t t') := by
induction t' with
| var n =>
  dsimp [substTerm]
  split_ifs with h_vn
  · exact NotOccur_of_isClosed t h_closed w
  · exact h_occ
| const c => exact NotOccur.const _
| henkin c => exact NotOccur.henkin _
| zero => exact NotOccur.zero
| succ t'' ih =>
  cases h_occ with | succ _ h =>
  dsimp [substTerm]
  exact NotOccur.succ _ (ih h)
| add t1 t2 ih1 ih2 =>
  cases h_occ with | add _ _ h1 h2 =>
  dsimp [substTerm]
  exact NotOccur.add _ _ (ih1 h1) (ih2 h2)
| mul t1 t2 ih1 ih2 =>
  cases h_occ with | mul _ _ h1 h2 =>
  dsimp [substTerm]
  exact NotOccur.mul _ _ (ih1 h1) (ih2 h2)
| exp t1 t2 ih1 ih2 =>
  cases h_occ with | exp _ _ h1 h2 =>
  dsimp [substTerm]
  exact NotOccur.exp _ _ (ih1 h1) (ih2 h2)
| pred t'' ih =>
  cases h_occ with | pred _ h =>
```

```
  dsimp [substTerm]
  exact NotOccur.pred _ (ih h)


--
============================================================================
-- 4. CORRECCIÓN EN NotFreeForm_substForm
--
============================================================================

lemma NotFreeForm_substForm {s : Nat} (φ : ArithForm s) (v : Nat) (w : Nat) (t :
ArithTerm s)
  (h_free : NotFreeForm w φ) (h_closed : isClosed t = true) :
  NotFreeForm w (substForm v t φ) := by
 induction φ generalizing v with
 | bottom => any_goals trivial
 | eq t1 t2 =>
  rw [substForm_eq]
  unfold NotFreeForm at h_free ⊢
  exact ⟨NotOccur_subst_other t1 v w t h_free.1 (NotOccur_of_isClosed t h_closed w),
NotOccur_subst_other t2 v w t h_free.2 (NotOccur_of_isClosed t h_closed w)⟩
 | impl f1 f2 ih1 ih2 =>
  rw [substForm_impl]
  unfold NotFreeForm at h_free ⊢
  exact ⟨ih1 v h_free.1, ih2 v h_free.2⟩
 | neg f ih =>
  rw [substForm_neg]
  unfold NotFreeForm at h_free ⊢
  exact ih v h_free
 | forall_form n f ih =>
```

```
  by_cases h_vn : v = n
  · rw [substForm_forall_form, if_pos h_vn]
    by_cases h_wn : w = n
    · unfold NotFreeForm
      rw [if_pos h_wn]
      trivial
    · unfold NotFreeForm
      rw [if_neg h_wn]
      have h_unfold := h_free
      unfold NotFreeForm at h_unfold
      rw [if_neg h_wn] at h_unfold
      exact h_unfold
  · rw [substForm_forall_form, if_neg h_vn]
    by_cases h_wn : w = n
    · unfold NotFreeForm
      rw [if_pos h_wn]
      trivial
    · unfold NotFreeForm
      rw [if_neg h_wn]
      have h_free' : NotFreeForm w f := by
        have h_unfold := h_free
        unfold NotFreeForm at h_unfold
        rw [if_neg h_wn] at h_unfold
        exact h_unfold
      exact ih v h_free'
| provable_op h t' =>
  rw [substForm_provable_op]
  unfold NotFreeForm at h_free ⊢
```

```
  exact NotOccur_subst_other t' v w t h_free (NotOccur_of_isClosed t h_closed w)
| true_op h t' =>
  rw [substForm_true_op]
  unfold NotFreeForm at h_free ⊢
  exact NotOccur_subst_other t' v w t h_free (NotOccur_of_isClosed t h_closed w)
| list_nth_eq t1 t2 t3 =>
  rw [substForm_list_nth_eq]
  unfold NotFreeForm at h_free ⊢
  exact ⟨NotOccur_subst_other t1 v w t h_free.1 (NotOccur_of_isClosed t h_closed w), NotOccur_subst_other t2 v w t h_free.2.1 (NotOccur_of_isClosed t h_closed w), NotOccur_subst_other t3 v w t h_free.2.2 (NotOccur_of_isClosed t h_closed w)⟩
```

```
lemma NotFreeInSet_substForm {s : Nat} (v : Nat) (w : Nat) (t : TermClosed s) (Γ : Set (ArithForm s))
  (h_free : NotFreeInSet w Γ) :
  NotFreeInSet w {substForm v t.val ψ | ψ ∈ Γ} := by
intro ψ' hψ'
rcases hψ' with ⟨ψ, hψ, rfl⟩
exact NotFreeForm_substForm ψ v w t.val (h_free ψ hψ) t.property
```

Lemas de simplificación de sustitución y distribución sobre conjuntos

Se demuestran dos lemas que completan las herramientas de manipulación sintáctica: la invarianza de una fórmula bajo sustitución cuando la variable no es libre, y la distribución de la sustitución sobre la operación de inserción en conjuntos.

Lema `substForm_fresh_of_NotFreeForm`: Invarianza bajo sustitución de variable no libre

Establece que si una variable v no es libre en una fórmula φ, entonces sustituir v por cualquier término t deja φ inalterada:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \text{NotFreeForm } v\ \varphi \implies \text{substForm } v\ t\ \varphi = \varphi $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos con términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`):** `NotFreeForm v φ` implica `NotOccur v` en cada subtérmino. Se aplica `substTerm_fresh` a cada uno, que establece que la sustitución no modifica el término.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción a las subfórmulas.

- **`forall_form n f`:** Si $v = n$, la sustitución no penetra el cuantificador (por definición de `substForm`), y la fórmula queda igual. Si $v \neq n$, la hipótesis `NotFreeForm v (forall_form n f)` implica que `NotFreeForm v f` (ya que $v \neq n$, la condición `if_neg` da esta conclusión). Se aplica la hipótesis de inducción al cuerpo f.

Significado

Este lema es una generalización de `substForm_fresh` (que requería la condición más fuerte `NotOccurForm`). La diferencia es sutil pero importante: `NotFreeForm` permite que la variable aparezca ligada, mientras que `NotOccurForm` exige que no aparezca en absoluto. Para la invarianza bajo sustitución, es suficiente que la variable no sea libre, ya que la sustitución solo afecta a las ocurrencias libres. Este lema es utilizado profusamente en las simplificaciones de `NumForm` y otras fórmulas complejas, donde se necesita garantizar que ciertas sustituciones no alteran subfórmulas donde la variable aparece ligada.

Lema `substFormSet_insert`: Distribución de la sustitución sobre la inserción

Establece la propiedad de conmutación entre la sustitución y la operación de inserción en conjuntos:

$$ \{\text{substForm } v\ t\ \chi \mid \chi \in \text{insert } \varphi\ \Gamma\} = \text{insert } (\text{substForm } v\ t\ \varphi)\ \{\text{substForm } v\ t\ \chi \mid \chi \in \Gamma\} $$

La demostración es por extensionalidad, analizando los casos de pertenencia a la inserción. Es el análogo para `substForm` del lema `substHenkinSet_insert` visto anteriormente.

Significado

Este lema de distribución es una herramienta técnica para la manipulación de contextos deductivos bajo sustituciones. Permite "empujar" la sustitución a través de las operaciones de construcción de conjuntos, lo cual es necesario en las

demostraciones de preservación de la demostrabilidad bajo transformaciones sintácticas (como la eliminación de constantes de Henkin). La igualdad establece que la imagen bajo sustitución de un conjunto extendido con φ es exactamente la extensión del conjunto imagen con la fórmula sustituida.

Junto con `substForm_fresh_of_NotFreeForm`, estos lemas completan el conjunto de herramientas para la manipulación de fórmulas con variables ligadas y libres, permitiendo simplificar expresiones complejas y razonar sobre la estructura de los conjuntos de fórmulas transformados por operaciones sintácticas. Son utilizados en las demostraciones de los teoremas de preservación de frescura en derivaciones y en la construcción del modelo canónico.

```
--
==========================================================================

-- 5. CORRECCIÓN EN substForm_fresh_of_NotFreeForm

--
==========================================================================




lemma substForm_fresh_of_NotFreeForm {s : Nat} (φ : ArithForm s) (v : Nat) (t :
ArithTerm s) (h : NotFreeForm v φ) :

   substForm v t φ = φ := by

 induction φ generalizing v with
```

```
| bottom => rfl
| eq t1 t2 =>
  unfold NotFreeForm at h
  rw [substForm_eq]
  rw [substTerm_fresh t1 v t h.1, substTerm_fresh t2 v t h.2]
| impl f1 f2 ih1 ih2 =>
  unfold NotFreeForm at h
  rw [substForm_impl]
  rw [ih1 v h.1, ih2 v h.2]
| neg f ih =>
  unfold NotFreeForm at h
  rw [substForm_neg]
  rw [ih v h]
| forall_form n f ih =>
  unfold NotFreeForm at h
  rw [substForm_forall_form]
  by_cases h_vn : v = n
  · rw [if_pos h_vn]
  · rw [if_neg h_vn]
    rw [ih v (by rwa [if_neg h_vn] at h)]
| provable_op h t' =>
  unfold NotFreeForm at h
  rw [substForm_provable_op]
  rw [substTerm_fresh t' v t h]
| true_op h t' =>
  unfold NotFreeForm at h
  rw [substForm_true_op]
  rw [substTerm_fresh t' v t h]
```

```
  | list_nth_eq t1 t2 t3 =>
    unfold NotFreeForm at h
    rw [substForm_list_nth_eq]
    rw [substTerm_fresh t1 v t h.1, substTerm_fresh t2 v t h.2.1, substTerm_fresh t3 v t
h.2.2]



lemma substFormSet_insert {s : Nat} (v : Nat) (t : ArithTerm s) (φ : ArithForm s) (Γ :
Set (ArithForm s)) :
    {χ | ∃ χ' ∈ insert φ Γ, substForm v t χ' = χ} = insert (substForm v t φ) {χ | ∃ χ' ∈ Γ,
substForm v t χ' = χ} := by
    ext χ
    simp only [Set.mem_insert_iff, Set.mem_ofPred_eq]
    constructor
    · rintro ⟨χ', h_or, rfl⟩
      cases h_or with
      | inl h1 => left; rw [h1]
      | inr h2 => right; exact ⟨χ', h2, rfl⟩
    · rintro (rfl | ⟨χ', h_χ', rfl⟩)
      · exact ⟨φ, Or.inl rfl, rfl⟩
      · exact ⟨χ', Or.inr h_χ', rfl⟩
```

Lemas de preservación de `IsFreeFor` en conjuntos y bajo auto-sustitución

Se demuestran dos lemas que extienden las propiedades de la condición de libertad (`IsFreeFor`) a contextos de conjuntos de fórmulas y a operaciones de sustitución sobre la misma variable.

Lema `IsFreeFor_insert`: Preservación de libertad en inserciones

Establece que si un término cerrado t es libre para una variable v en una fórmula φ y también en todas las fórmulas de un conjunto Γ, entonces es libre para v en todas las fórmulas del conjunto extendido $\Gamma \cup \{\varphi\}$:

$$ \forall v \in \mathbb{N},\ \forall t \in \text{TermClosed } s,\ \forall \varphi \in \text{ArithForm } s,\ \forall \Gamma \subseteq \text{ArithForm } s, $$

$$ \text{IsFreeFor } v\ t.\text{val}\ \varphi \land (\forall \psi \in \Gamma,\ \text{IsFreeFor } v\ t.\text{val}\ \psi) \implies \forall \psi \in \text{insert } \varphi\ \Gamma,\ \text{IsFreeFor } v\ t.\text{val}\ \psi $$

La demostración es por análisis de casos sobre la pertenencia a la inserción: si $\psi = \varphi$, se usa la primera hipótesis; si $\psi \in \Gamma$, se usa la segunda.

Significado

Este lema es una herramienta auxiliar para construcciones inductivas sobre conjuntos de fórmulas. Cuando se construye una teoría extendida añadiendo fórmulas una a una (como en la secuencia de Lindenbaum), es necesario garantizar que ciertas condiciones de libertad se mantienen para todas las fórmulas del conjunto. Este lema permite verificar que la condición se preserva al añadir una nueva fórmula.

Lema `IsFreeFor_subst_same`: Preservación de libertad bajo auto-sustitución

Establece que si un término u es libre para una variable w en una fórmula φ, entonces al sustituir w por un término cerrado t dentro de u, el término resultante sigue siendo libre para w en φ:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall w \in \mathbb{N},\ \forall t \in \text{TermClosed } s,\ \forall u \in \text{ArithTerm } s, $$

$$ \text{IsFreeFor } w\ u\ \varphi \implies \text{IsFreeFor } w\ (\text{substTerm } w\ t.\text{val}\ u)\ \varphi $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos sin cuantificadores:** `IsFreeFor` no depende de u, por lo que la conclusión es trivial.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción a las subfórmulas.

- **`forall_form n f`:** Si $w = n$, la sustitución no penetra y la condición es trivial. Si $w \neq n$, la hipótesis `IsFreeFor w u (forall_form n f)` proporciona `NotOccur n u` e `IsFreeFor w u f`. Se debe verificar que `NotOccur n (substTerm w t.val u)`. Por el lema `NotOccur_substTerm`, como $n \neq w$ (ya que $w \neq n$) y n no ocurre en u, la sustitución de w por $t.\text{val}$ (que es cerrado y por tanto no contiene a n) preserva que n no ocurre en el resultado. La hipótesis de inducción proporciona la preservación de la libertad en f.

Significado

Este lema aborda una situación sutil: cuando se tiene un término u que es libre para w en φ, y se modifica u sustituyendo precisamente w por un término

cerrado, el nuevo término sigue siendo libre para w en φ. La clave es que la sustitución elimina las ocurrencias de w en u (reemplazándolas por $t.\text{val}$, que es cerrado y por tanto no contiene variables libres), lo que no puede incrementar el riesgo de captura. De hecho, la condición de libertad debería ser "más fácil" de satisfacer después de la sustitución, ya que las posibles ocurrencias de w en u que podrían interactuar con cuantificadores han sido eliminadas.

Este lema es utilizado en contextos donde se aplican múltiples sustituciones sucesivas y se necesita verificar que las condiciones de aplicabilidad de `forall_elim` se mantienen a lo largo de transformaciones sintácticas. Junto con `IsFreeFor_subst_term` (que trata la sustitución sobre una variable distinta), completa el cuadro de cómo la propiedad `IsFreeFor` se comporta bajo los dos tipos de modificación de términos: sustitución sobre la misma variable y sobre variable distinta.

```
lemma IsFreeFor_insert {s : Nat} (v : Nat) (t : TermClosed s) (φ : ArithForm s) (Γ : Set (ArithForm s))
   (h_φ : IsFreeFor v t.val φ) (h_Γ : ∀ ψ ∈ Γ, IsFreeFor v t.val ψ) :
   ∀ ψ ∈ insert φ Γ, IsFreeFor v t.val ψ := by
 intro ψ hψ
 cases hψ with
 | inl h => subst h; exact h_φ
 | inr h => exact h_Γ ψ h
```

```
lemma IsFreeFor_subst_same {s : Nat} (φ : ArithForm s) (w : Nat) (t : TermClosed s) (u
: ArithTerm s)
    (h_free : IsFreeFor w u φ) :
    IsFreeFor w (substTerm w t.val u) φ := by
  induction φ generalizing w u with
  | bottom => trivial
  | eq _ _ => trivial
  | impl f1 f2 ih1 ih2 =>
    dsimp [IsFreeFor] at h_free ⊢
    exact ⟨ih1 w u h_free.1, ih2 w u h_free.2⟩
  | neg f ih =>
    dsimp [IsFreeFor] at h_free ⊢
    exact ih w u h_free
  | forall_form n f ih =>
    dsimp [IsFreeFor] at h_free ⊢
    by_cases h_wn : w = n
    · rw [if_pos h_wn]
      trivial
    · rw [if_neg h_wn] at h_free ⊢
      exact ⟨NotOccur_substTerm u w n t.val h_free.1 t.property, ih w u h_free.2⟩
  | provable_op _ _ => trivial
  | true_op _ _ => trivial
  | list_nth_eq _ _ _ => trivial
```

Lema `IsFreeFor_subst_lemma`: Preservación de libertad bajo sustitución simultánea

Se demuestra un lema fundamental que establece la preservación de la condición de libertad cuando tanto la fórmula como el término de sustitución son transformados por una misma sustitución sobre una variable distinta.

Enunciado formal

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \forall t \in \text{TermClosed } s,\ \forall u \in \text{ArithTerm } s, $$

$$ \text{IsFreeFor } w\ u\ \varphi \implies \text{IsFreeFor } w\ (\text{substTerm } v\ t.\text{val}\ u)\ (\text{substForm } v\ t.\text{val}\ \varphi) $$

Es decir, si u es libre para w en φ, entonces al aplicar una sustitución $[v := t]$ (con t cerrado) tanto a u como a φ, el término resultante sigue siendo libre para w en la fórmula resultante.

Estructura de la demostración

Por inducción sobre φ:

- **Casos sin cuantificadores:** `IsFreeFor` no depende de los términos, por lo que la conclusión es trivial.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción a las subfórmulas, y la sustitución se distribuye sobre los conectivos.

- **`forall_form n f`:** Se analiza mediante casos sobre si $v = n$ y si $w = n$:

  - **Si $v = n$:** La sustitución no penetra el cuantificador. Si además $w = n$, `IsFreeFor` es trivial (la variable está ligada). Si $w \neq n$, la hipótesis `IsFreeFor w u (forall_form n f)` proporciona `NotOccur n u` e `IsFreeFor w u f`. Se debe verificar:

    - `NotOccur n (substTerm v t.val u)`: Como $v = n$ y t es cerrado (no contiene a n), por `NotOccur_substTerm` la propiedad se preserva (o alternativamente, `NotOccur_subst_other` con $n \neq w$).

    - `IsFreeFor w (substTerm v t.val u) f`: Se sigue del lema `IsFreeFor_subst_term` aplicado a f.

  - **Si $v \neq n$:** La sustitución atraviesa el cuantificador. Los mismos razonamientos se aplican, con la diferencia de que la sustitución se propaga al cuerpo f. La hipótesis de inducción se aplica a f con los mismos v, w, u, y la condición `NotOccur n (substTerm v t.val u)` se verifica como antes.

Significado

`IsFreeFor_subst_lemma` es uno de los lemas más importantes para la manipulación de fórmulas con sustituciones múltiples. Captura la idea de que la condición de "ser libre para" es estable bajo transformaciones sintácticas uniformes: si se transforma simultáneamente la fórmula y el término de sustitución mediante una misma operación (que no interfiere con la variable crítica w), la propiedad de libertad se preserva.

Este lema es utilizado en la demostración de `substHenkin_preservation_of_fresh` (el teorema de eliminación de constantes de Henkin), donde se necesita garantizar que después de reemplazar una constante de Henkin por una variable fresca en toda una derivación, las aplicaciones de `forall_elim` en la derivación transformada siguen siendo válidas. La transformación afecta tanto a la fórmula cuantificada como al

término que se sustituye, y `IsFreeFor_subst_lemma` asegura que la condición de libertad se mantiene a través de esta transformación.

La distinción de casos $v = n$ versus $v \neq n$ refleja si la sustitución externa interfiere con la variable ligada del cuantificador. En ambos casos, la clave es que t es un término cerrado, por lo que no contiene variables que pudieran ser capturadas, y la preservación de `NotOccur n` en el término transformado está garantizada por los lemas de frescura bajo sustitución. Esta observación —que los términos cerrados son "inocuos" respecto a las condiciones de libertad— es recurrente en todo el desarrollo y es una de las razones por las que la construcción de Henkin insiste en que los testigos sean términos cerrados.

```
--
==========================================================================

-- 6. CORRECCIÓN EN IsFreeFor_subst_lemma (Evita colisión de nombres)

--
==========================================================================


lemma IsFreeFor_subst_lemma {s : Nat} (φ : ArithForm s) (v w : Nat) (t : TermClosed
s) (u : ArithTerm s)

   (h_free : IsFreeFor w u φ) :

   IsFreeFor w (substTerm v t.val u) (substForm v t.val φ) := by

 unfold TermClosed at t

 induction φ generalizing v w u with

 | bottom => trivial

 | eq _ _ => trivial

 | impl f1 f2 ih1 ih2 =>

   rw [substForm_impl]
```

```
    dsimp [IsFreeFor] at h_free ⊢
    exact ⟨ih1 v w u h_free.1, ih2 v w u h_free.2⟩
  | neg f ih =>
    rw [substForm_neg]
    dsimp [IsFreeFor] at h_free ⊢
    exact ih v w u h_free
  | forall_form n f ih =>
    rw [substForm_forall_form]
    by_cases h_vn : v = n
    · rw [if_pos h_vn]
      by_cases h_wn : w = n
      · dsimp [IsFreeFor]
        rw [if_pos h_wn]
        trivial
      · dsimp [IsFreeFor] at h_free ⊢
        rw [if_neg h_wn] at h_free ⊢
        exact ⟨NotOccur_subst_other u v n t.val h_free.1 (NotOccur_of_isClosed t.val
t.property n), IsFreeFor_subst_term f v w t u h_free.2⟩
    · rw [if_neg h_vn]
      by_cases h_wn : w = n
      · dsimp [IsFreeFor]
        rw [if_pos h_wn]
        trivial
      · dsimp [IsFreeFor] at h_free ⊢
        rw [if_neg h_wn] at h_free ⊢
        exact ⟨NotOccur_subst_other u v n t.val h_free.1 (NotOccur_of_isClosed t.val
t.property n), ih v w u h_free.2⟩
  | provable_op h t' => trivial
  | true_op h t' => trivial
```

```
  | list_nth_eq t1 t2 t3 => trivial
```

Teorema `subst_derivation_fresh`: Preservación de la demostrabilidad bajo sustitución fresca

Se demuestra el teorema fundamental que establece que la demostrabilidad se preserva bajo sustituciones de variables por términos cerrados, siempre que la variable sustituida sea fresca respecto a la derivación original.

Enunciado formal

$$ \forall \Gamma \subseteq \text{ArithForm } s,\ \forall \varphi \in \text{ArithForm } s,\ \forall d : \Gamma \vdash \varphi, $$

$$ \forall v \in \mathbb{N},\ \forall t \in \text{TermClosed } s,\ \text{FreshForDerivation } v\ d \implies $$

$$ \{\text{substForm } v\ t.\text{val}\ \psi \mid \psi \in \Gamma\} \vdash \text{substForm } v\ t.\text{val}\ \varphi $$

Estructura de la demostración

La prueba procede por inducción sobre la derivación de `FreshForDerivation v d`, reconstruyendo una derivación en el sistema transformado para cada regla de inferencia:

Reglas que no involucran variables ligadas ni sustituciones (`hyp`, `imp_elim`, `double_neg`, `neg_elim`, reglas de congruencia de igualdad, `weaken`, `ax_peano1`, `ax_peano2`): Se aplica la misma regla a las derivaciones transformadas de las premisas, distribuyendo la sustitución sobre las fórmulas involucradas. Para las reglas de congruencia de la igualdad, la sustitución se propaga a los términos mediante `substTerm`.

Reglas que modifican el contexto (`imp_intro`, `neg_intro`): Se utiliza `substFormSet_insert` para distribuir la sustitución sobre la inserción de la fórmula descargada en el contexto.

`forall_intro`: Requiere verificar que la variable generalizada w' sigue sin ser libre en el contexto transformado. Esto se sigue del lema `NotFreeInSet_substForm`, que garantiza que la sustitución de v por t (con t cerrado) no introduce nuevas ocurrencias libres de w'.

`forall_elim`: Es uno de los casos más técnicos. Se debe aplicar `forall_elim` en el sistema transformado con la misma variable w' y el mismo término t'. Para ello se necesita:

1. La derivación transformada de la premisa $\forall w'\ \varphi'$, que la hipótesis de inducción proporciona como $\forall w'\ (\text{substForm } v\ t.\text{val}\ \varphi')$ (ya que $v \neq w'$ por la condición de frescura, la sustitución atraviesa el cuantificador).

2. La preservación de la condición `IsFreeFor w' t' (substForm v t.val φ')`. Esto se sigue del lema `IsFreeFor_subst_lemma`, que garantiza que si t' era libre para w' en φ', entonces $\text{substTerm } v\ t.\text{val}\ t'$ es libre para w' en $\text{substForm } v\ t.\text{val}\ \varphi'$. Pero como v no ocurre en t' (por la condición `h_vt'` de `FreshForDerivation`), $\text{substTerm } v\ t.\text{val}\ t' = t'$.

3. La conmutación de las sustituciones: $\text{substForm } v\ t.\text{val}\ (\text{substForm } w'\ t'\ \varphi') = \text{substForm } w'\ t'\ (\text{substForm } v\ t.\text{val}\ \varphi')$, que se sigue de `substForm_comm` (ya que $v \neq w'$, w' no ocurre en $t.\text{val}$ por ser cerrado, y v no ocurre en t').

`eq_subst`: Similar a `forall_elim` en complejidad. Requiere conmutar las sustituciones y preservar las condiciones `IsFreeFor` para ambos términos t_1 y t_2.

`induction`: Requiere conmutar las sustituciones con `ArithTerm.zero` y con `ArithTerm.succ (ArithTerm.var w')`, y verificar que las hipótesis de inducción se aplican a las derivaciones transformadas.

Significado

`subst_derivation_fresh` es uno de los teoremas más importantes del desarrollo. Establece que la demostrabilidad es invariante bajo sustituciones de variables por términos cerrados, siempre que la variable sustituida sea completamente fresca respecto a la derivación. Esta propiedad es fundamental para:

1. **Eliminación de constantes de Henkin:** Cuando se reemplaza una constante de Henkin por una variable fresca, `subst_derivation_fresh` garantiza que la derivación transformada sigue siendo válida (es el análogo para variables de `substHenkin_preservation_of_fresh`).

2. **Generalización universal:** Permite instanciar variables ligadas con términos cerrados arbitrarios, preservando la estructura deductiva.

3. **Construcción del modelo canónico:** En la definición de `satisfies`, las fórmulas se evalúan bajo asignaciones de términos cerrados. `subst_derivation_fresh` asegura que las equivalencias deductivas se mantienen bajo estas asignaciones.

La demostración es extensa pero sistemática: cada regla de inferencia se transforma en la regla correspondiente en el sistema transformado, utilizando los lemas de conmutación y preservación de frescura desarrollados previamente. La condición `FreshForDerivation v d` es la que garantiza que todas estas transformaciones son lícitas: v no interfiere con ninguna variable ligada, no aparece en ningún término ni

fórmula de la derivación, y por tanto puede ser sustituida sin riesgo de captura o violación de condiciones de libertad.

```
--
================================================================

-- 7. TEOREMA PRINCIPAL (subst_derivation_fresh)

--
================================================================


theorem subst_derivation_fresh {s : Nat} {Γ : Set (ArithForm s)} {φ : ArithForm s}

   (v : Nat) (t : TermClosed s)

   (h : SPA_Deduction_Internal s Γ φ)

   (h_fresh : FreshForDerivation v h) :

   SPA_Deduction_Internal s {ψ | ∃ ψ' ∈ Γ, substForm v t.val ψ' = ψ} (substForm v t.val
φ) := by

 unfold TermClosed at t

 induction h_fresh with

 | hyp Γ' φ' h_in h_w =>

   apply SPA_Deduction_Internal.hyp

   exact ⟨φ', h_in, rfl⟩


 | imp_intro Γ' φ' ψ' h_sub h_wφ ih_fresh ih =>

   rw [substFormSet_insert] at ih

   rw [substForm_impl]
```

```
  apply SPA_Deduction_Internal.imp_intro
  exact ih

| imp_elim Γ' φ' ψ' h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_impl] at ih1
  apply SPA_Deduction_Internal.imp_elim (φ := substForm v t.val φ')
  · exact ih1
  · exact ih2

| double_neg Γ' φ' h ih_fresh ih =>
  rw [substForm_neg] at ih
  rw [substForm_neg] at ih
  apply SPA_Deduction_Internal.double_neg
  exact ih

| neg_intro Γ' φ' h h_wφ ih_fresh ih =>
  rw [substFormSet_insert] at ih
  rw [substForm_neg]
  apply SPA_Deduction_Internal.neg_intro
  exact ih

| neg_elim Γ' φ' h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_neg] at ih2
  apply SPA_Deduction_Internal.neg_elim (φ := substForm v t.val φ')
  · exact ih1
  · exact ih2

| forall_intro Γ' w' φ' h_sub h_w'Γ hnw ih_fresh ih =>
```

```
    rw [substForm_forall_form, if_neg hnw.symm]
    apply SPA_Deduction_Internal.forall_intro
    · exact ih
    · exact NotFreeInSet_substForm v w' t Γ' h_w'Γ

  | forall_elim Γ' w' φ' t' h_sub h_free hnw h_vt' ih_fresh ih =>
    have h_comm : substForm v t.val (substForm w' t' φ') = substForm w' t' (substForm v t.val φ') := by
      have h_not_occur : NotOccur w' t.val := NotOccur_of_isClosed t.val t.property w'
      rw [substForm_comm φ' v w' t.val t' hnw.symm h_not_occur h_vt']
      have h_fresh_term : substTerm v t.val t' = t' := substTerm_fresh t' v t.val h_vt'
      rw [h_fresh_term]
    rw [h_comm]
    rw [substForm_forall_form, if_neg hnw.symm] at ih
    apply SPA_Deduction_Internal.forall_elim (v := w') (φ := substForm v t.val φ') (t := t')
    · exact ih
    · have h_is_free := IsFreeFor_subst_lemma φ' v w' t t' h_free
      have h_fresh_term : substTerm v t.val t' = t' := substTerm_fresh t' v t.val h_vt'
      rw [h_fresh_term] at h_is_free
      exact h_is_free

  | eq_refl Γ' t' h_vt' =>
    rw [substForm_eq]
    apply SPA_Deduction_Internal.eq_refl

  | eq_symm Γ' t1 t2 h' ih_fresh ih =>
    rw [substForm_eq] at ih
    rw [substForm_eq]
```

```
  apply SPA_Deduction_Internal.eq_symm
  exact ih

| eq_trans Γ' t1 t2 t3 h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_eq] at ih1 ih2
  rw [substForm_eq]
  apply SPA_Deduction_Internal.eq_trans (t2 := substTerm v t.val t2)
  · exact ih1
  · exact ih2

| eq_succ Γ' t1 t2 h' ih_fresh ih =>
  rw [substForm_eq] at ih
  rw [substForm_eq]
  apply SPA_Deduction_Internal.eq_succ
  exact ih

| eq_add Γ' t1 t2 u1 u2 h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_eq] at ih1 ih2
  rw [substForm_eq]
  apply SPA_Deduction_Internal.eq_add
  · exact ih1
  · exact ih2

| eq_mul Γ' t1 t2 u1 u2 h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_eq] at ih1 ih2
  rw [substForm_eq]
  apply SPA_Deduction_Internal.eq_mul
  · exact ih1
```

```
  · exact ih2

| eq_exp Γ' t1 t2 u1 u2 h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_eq] at ih1 ih2
  rw [substForm_eq]
  apply SPA_Deduction_Internal.eq_exp
  · exact ih1
  · exact ih2

| eq_pred Γ' t1 t2 h' ih_fresh ih =>
  rw [substForm_eq] at ih
  rw [substForm_eq]
  apply SPA_Deduction_Internal.eq_pred
  exact ih

| eq_provable_op Γ' h_eq t1 t2 h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_eq] at ih1
  rw [substForm_provable_op] at ih2
  rw [substForm_provable_op]
  apply SPA_Deduction_Internal.eq_provable_op (s_l := _) (h_eq := h_eq)
  · exact ih1
  · exact ih2

| eq_true_op Γ' h_eq t1 t2 h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_eq] at ih1
  rw [substForm_true_op] at ih2
  rw [substForm_true_op]
  apply SPA_Deduction_Internal.eq_true_op (s_l := _) (h_eq := h_eq)
```

```
  · exact ih1
  · exact ih2

| eq_list_nth_eq_1 Γ' t1 t2 t3 u1 h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_eq] at ih1
  rw [substForm_list_nth_eq] at ih2
  rw [substForm_list_nth_eq]
  apply SPA_Deduction_Internal.eq_list_nth_eq_1
  · exact ih1
  · exact ih2

| eq_list_nth_eq_2 Γ' t1 t2 t3 u2 h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_eq] at ih1
  rw [substForm_list_nth_eq] at ih2
  rw [substForm_list_nth_eq]
  apply SPA_Deduction_Internal.eq_list_nth_eq_2
  · exact ih1
  · exact ih2

| eq_list_nth_eq_3 Γ' t1 t2 t3 u3 h1 h2 ih1_fresh ih2_fresh ih1 ih2 =>
  rw [substForm_eq] at ih1
  rw [substForm_list_nth_eq] at ih2
  rw [substForm_list_nth_eq]
  apply SPA_Deduction_Internal.eq_list_nth_eq_3
  · exact ih1
  · exact ih2
```

```
 | eq_subst Γ' w' φ' t1 t2 h1 h2 h_free1 h_free2 hnw h_wφ ih1_fresh ih2_fresh ih1 ih2
=>

  have h_vt12 := FreshForDerivation_imp_NotOccurForm v h1 ih1_fresh

  unfold NotOccurForm at h_vt12

  have h_vt1 : NotOccur v t1 := h_vt12.1

  have h_vt2 : NotOccur v t2 := h_vt12.2

  have h_comm1 : substForm v t.val (substForm w' t1 φ') = substForm w' (substTerm
v t.val t1) (substForm v t.val φ') :=

   substForm_comm φ' v w' t.val t1 hnw.symm (NotOccur_of_isClosed t.val t.property
w') h_vt1

  have h_comm2 : substForm v t.val (substForm w' t2 φ') = substForm w' (substTerm
v t.val t2) (substForm v t.val φ') :=

   substForm_comm φ' v w' t.val t2 hnw.symm (NotOccur_of_isClosed t.val t.property
w') h_vt2

  rw [h_comm1] at ih2

  rw [h_comm2]

  rw [substForm_eq] at ih1

  apply SPA_Deduction_Internal.eq_subst (v := w') (φ := substForm v t.val φ') (t1 :=
substTerm v t.val t1) (t2 := substTerm v t.val t2)

  · exact ih1

  · exact ih2

  · exact IsFreeFor_subst_lemma φ' v w' t t1 h_free1

  · exact IsFreeFor_subst_lemma φ' v w' t t2 h_free2


 | weaken Γ' φ' Γ'' h' h_sub ih_fresh ih =>

  apply SPA_Deduction_Internal.weaken (Γ := {ψ | ∃ ψ' ∈ Γ', substForm v t.val ψ' = ψ})

  · exact ih

  · intro ψ h_ψ

   rcases h_ψ with ⟨ψ', hψ', rfl⟩

   exact ⟨ψ', h_sub hψ', rfl⟩
```

```
  | ax_peano1 Γ' x' h_vx =>
    rw [substForm_neg, substForm_eq]
    apply SPA_Deduction_Internal.ax_peano1

  | ax_peano2 Γ' x' y' h_vx h_vy =>
    rw [substForm_impl, substForm_eq, substForm_eq]
    apply SPA_Deduction_Internal.ax_peano2

  | induction Γ' φ' w' h1 h2 hnw h_wφ ih1_fresh ih2_fresh ih1 ih2 =>
    have h_comm1 : substForm v t.val (substForm w' ArithTerm.zero φ') = substForm w' ArithTerm.zero (substForm v t.val φ') := by
      have h_comm_raw := substForm_comm φ' v w' t.val ArithTerm.zero hnw.symm (NotOccur_of_isClosed t.val t.property w') NotOccur.zero
      have h_fresh_term : substTerm v t.val ArithTerm.zero = ArithTerm.zero := rfl
      rw [h_comm_raw, h_fresh_term]
    have h_comm2 : substForm v t.val (substForm w' (ArithTerm.succ (ArithTerm.var w' : ArithTerm s)) φ') = substForm w' (ArithTerm.succ (ArithTerm.var w' : ArithTerm s)) (substForm v t.val φ') := by
      have h_not_occur_vw : NotOccur v (ArithTerm.succ (ArithTerm.var w' : ArithTerm s)) := NotOccur.succ _ (NotOccur.var w' hnw.symm)
      have h_comm_raw := substForm_comm φ' v w' t.val (ArithTerm.succ (ArithTerm.var w' : ArithTerm s)) hnw.symm (NotOccur_of_isClosed t.val t.property w') h_not_occur_vw
      have h_fresh_term : substTerm v t.val (ArithTerm.succ (ArithTerm.var w' : ArithTerm s)) = ArithTerm.succ (ArithTerm.var w') := by
        dsimp [substTerm]
        rw [if_neg hnw.symm]
      rw [h_comm_raw, h_fresh_term]
    rw [h_comm1] at ih1
    rw [substForm_forall_form, if_neg hnw.symm] at ih2
```

```
  rw [substForm_impl] at ih2
  rw [h_comm2] at ih2
  rw [substForm_forall_form, if_neg hnw.symm]
  apply SPA_Deduction_Internal.induction (φ := substForm v t.val φ') (v := w')
  · exact ih1
  · exact ih2

end SPA
```

Lemas de conmutación de sustituciones con variables

Se demuestran dos lemas que establecen propiedades de cancelación y conmutación para sustituciones que involucran variables frescas. Estos resultados son versiones generalizadas de lemas anteriores y son herramientas esenciales para la manipulación de fórmulas con múltiples variables ligadas.

Lema `substTerm_substTerm_var_comm`: Cancelación de sustituciones con variable intermedia

Establece que si una variable w no ocurre en un término t, entonces sustituir primero v por $\text{var } w$ y luego w por $\text{var } z$ es equivalente a sustituir directamente v por $\text{var } z$:

$$ \forall t \in \text{ArithTerm } s,\ \forall v, w, z \in \mathbb{N},\ \text{NotOccur } w\ t \implies $$

$$ \text{substTerm } w\ (\text{var } z)\ (\text{substTerm } v\ (\text{var } w)\ t) = \text{substTerm } v\ (\text{var } z)\ t $$

Estructura de la demostración

Por inducción sobre t:

- **`var n`:** Se analiza mediante casos:

  - Si $n = v$: La primera sustitución produce $\text{var } w$, la segunda la reemplaza por $\text{var } z$. El lado derecho produce directamente $\text{var } z$.

  - Si $n \neq v$: La primera sustitución no afecta. La segunda sustitución solo afectaría si $n = w$, pero `NotOccur w (var n)` fuerza $w \neq n$. Ambos lados preservan $\text{var } n$.

- **Casos recursivos:** Se invierte la hipótesis `NotOccur w t` y se aplican las hipótesis de inducción.

Significado

Este lema generaliza `substTerm_substTerm_self` (que era el caso $z = v$) y `substTerm_substTerm_henkin` (que era el caso con constante de Henkin). Muestra que la variable intermedia w actúa como un "puente" que puede ser eliminado: renombrar v a w y luego w a z es lo mismo que renombrar v a z directamente. Esta propiedad es fundamental para la alpha-conversión y el cambio de variables ligadas.

Lema `substForm_substForm_var_comm`: Cancelación en fórmulas

Extiende el resultado anterior al nivel de fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w, z \in \mathbb{N},\ \text{NotOccurForm } w\ \varphi \implies $$

$$ \text{substForm } w\ (\text{var } z)\ (\text{substForm } v\ (\text{var } w)\ \varphi) = \text{substForm } v\ (\text{var } z)\ \varphi $$

La demostración procede por casos sobre la estructura de φ (usando `match`), aplicando `substTerm_substTerm_var_comm` a los subtérminos y la hipótesis de inducción a las subfórmulas. El caso `forall_form n f` distingue si $v = n$ (la sustitución no penetra) o $v \neq n$, y utiliza que $w \neq n$ por `NotOccurForm w (forall_form n f)`.

Significado conjunto

Estos lemas son generalizaciones importantes de los resultados previos sobre conmutación de sustituciones. Mientras que `substForm_substForm_self` trataba el caso en que la variable intermedia se reemplaza por la variable original (cancelación total), y `substForm_substForm_henkin` trataba el reemplazo por una constante de Henkin, esta versión permite que la variable intermedia sea reemplazada por cualquier otra variable z. Esto proporciona una flexibilidad total para renombrar variables ligadas: se puede cambiar v por w (alpha-conversión) y luego w por z (otra alpha-conversión), y el resultado neto es equivalente a cambiar directamente v por z.

Esta propiedad de "transitividad" de la alpha-conversión es utilizada en los lemas de cambio de variable ligada (`change_var_equivalence_strong`) y en la construcción de testigos de Henkin, donde se necesita elegir variables frescas que no interfieran con las variables existentes. La capacidad de componer renombramientos de manera limpia es lo que permite manejar la complejidad de fórmulas con múltiples cuantificadores anidados, como `NumForm` y `BetaForm`. La condición `NotOccurForm w φ` es la hipótesis estándar de frescura que garantiza que la variable intermedia no interfiere con la estructura de la fórmula.

```
namespace SPA


--
============================================================================
-- SOPORTE DE COTAS Y CONMUTACIÓN PARA VARIABLES (ELIMINACIÓN DE AXIOMAS)
--
============================================================================


lemma substTerm_substTerm_var_comm {s : Nat} (t : ArithTerm s) (v w z : Nat) (hw : NotOccur w t) :
  substTerm w (ArithTerm.var z) (substTerm v (ArithTerm.var w) t) = substTerm v (ArithTerm.var z) t := by
 induction t with
 | var n =>
  simp only [substTerm_var]
  split_ifs with h1
  · subst h1
   simp only [substTerm_var]
   split_ifs
   · rfl
   · rfl
  · simp only [substTerm_var]
   split_ifs with h2
   · cases hw with | var _ h_ne => contradiction
```

```
    · rfl
  | const c' => rfl
  | henkin c' => rfl
  | zero => rfl
  | succ t' ih =>
    cases hw with | succ _ h_sub =>
    dsimp [substTerm]
    rw [ih h_sub]
  | add t1 t2 ih1 ih2 =>
    cases hw with | add _ _ h1 h2 =>
    dsimp [substTerm]
    rw [ih1 h1, ih2 h2]
  | mul t1 t2 ih1 ih2 =>
    cases hw with | mul _ _ h1 h2 =>
    dsimp [substTerm]
    rw [ih1 h1, ih2 h2]
  | exp t1 t2 ih1 ih2 =>
    cases hw with | exp _ _ h1 h2 =>
    dsimp [substTerm]
    rw [ih1 h1, ih2 h2]
  | pred t' ih =>
    cases hw with | pred _ h_sub =>
    dsimp [substTerm]
    rw [ih h_sub]


lemma substForm_substForm_var_comm {s : Nat} (φ : ArithForm s) (v w z : Nat) (hw :
NotOccurForm w φ) :
    substForm w (ArithTerm.var z) (substForm v (ArithTerm.var w) φ) = substForm v
(ArithTerm.var z) φ :=
```

```
  match φ with
  | ArithForm.bottom => rfl
  | ArithForm.eq t1 t2 => by
    have hw_left : NotOccur w t1 := hw.left
    have hw_right : NotOccur w t2 := hw.right
    rw [substForm_eq, substForm_eq]
    rw [substTerm_substTerm_var_comm t1 v w z hw_left, substTerm_substTerm_var_comm t2 v w z hw_right]
    rfl
  | ArithForm.impl f1 f2 => by
    have hw_left : NotOccurForm w f1 := hw.left
    have hw_right : NotOccurForm w f2 := hw.right
    rw [substForm_impl, substForm_impl]
    rw [substForm_substForm_var_comm f1 v w z hw_left, substForm_substForm_var_comm f2 v w z hw_right]
    rfl
  | ArithForm.neg f => by
    have hw_inner : NotOccurForm w f := hw
    rw [substForm_neg, substForm_neg]
    rw [substForm_substForm_var_comm f v w z hw_inner]
    rfl
  | ArithForm.forall_form n f => by
    have hw_left : w ≠ n := hw.left
    have hw_right : NotOccurForm w f := hw.right
    by_cases h_v : v = n
    · subst h_v
      rw [substForm_forall_form, if_pos rfl]
      rw [substForm_forall_form, if_neg hw_left]
      rw [substForm_fresh f w (ArithTerm.var z) hw_right]
```

```
    rw [substForm_forall_form, if_pos rfl]
  · rw [substForm_forall_form, if_neg h_v]
    rw [substForm_forall_form, if_neg hw_left]
    rw [substForm_substForm_var_comm f v w z hw_right]
    rw [substForm_forall_form, if_neg h_v]
| ArithForm.provable_op heq t => by
  have hw_inner : NotOccur w t := hw
  rw [substForm_provable_op, substForm_provable_op]
  rw [substTerm_substTerm_var_comm t v w z hw_inner]
  rfl
| ArithForm.true_op heq t => by
  have hw_inner : NotOccur w t := hw
  rw [substForm_true_op, substForm_true_op]
  rw [substTerm_substTerm_var_comm t v w z hw_inner]
  rfl
| ArithForm.list_nth_eq t1 t2 t3 => by
  have hw_t1 : NotOccur w t1 := hw.left
  have hw_t2 : NotOccur w t2 := hw.right.left
  have hw_t3 : NotOccur w t3 := hw.right.right
  rw [substForm_list_nth_eq, substForm_list_nth_eq]
  rw [substTerm_substTerm_var_comm t1 v w z hw_t1, substTerm_substTerm_var_comm t2 v w z hw_t2, substTerm_substTerm_var_comm t3 v w z hw_t3]
  rfl
```

Lemas de acotación para el máximo de variables bajo sustituciones

Se demuestran dos lemas que establecen cotas superiores para el máximo índice de variable en términos y fórmulas después de aplicar una sustitución. Estos resultados son fundamentales para garantizar que las operaciones sintácticas no introducen variables con índices arbitrariamente grandes.

Lema `max_var_term_substTerm`: Cota para términos tras sustitución

Establece que el máximo índice de variable en un término sustituido está acotado por el máximo entre el índice máximo del término original y el del término de sustitución:

$$ \forall t \in \text{ArithTerm } } s,\ \forall v \in \mathbb{N},\ \forall u \in \text{ArithTerm } } s, $$

$$ \text{max_var_term}(\text{substTerm } } v\ u\ t) \le \max(\text{max_var_term}(t), \text{max_var_term}(u)) $$

Estructura de la demostración

Por inducción sobre t:

- **`var n`:** Si $v = n$, el resultado es u, y su máximo es $\text{max_var_term}(u) \le \max(\ldots, \text{max_var_term}(u))$. Si $v \neq n$, el término no cambia, y $\text{max_var_term}(t) \le \max(\text{max_var_term}(t), \ldots)$.

- **Casos base (`const`, `henkin`, `zero`):** El máximo es 0, que es menor o igual que cualquier máximo.

- **Casos recursivos (`succ`, `pred`):** La cota se hereda de la hipótesis de inducción.

- **Casos binarios (`add`, `mul`, `exp`):** Se utiliza el lema `max_le_max_of_le` para combinar las cotas de los subtérminos.

Lema `max_var_form_substForm`: Cota para fórmulas tras sustitución

Extiende el resultado anterior al nivel de fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \forall u \in \text{ArithTerm } s, $$

$$ \text{max_var_form}(\text{substForm } v\ u\ \varphi) \le \max(\text{max_var_form}(\varphi), \text{max_var_term}(u)) $$

Estructura de la demostración

Por inducción sobre φ:

- **`bottom`:** El máximo es 0.

- **Casos con términos (`eq`, `provable_op`, `true_op`, `list_nth_eq`):** Se aplica `max_var_term_substTerm` a cada subtérmino y se combinan con `max_le_max_of_le`.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Si la sustitución no penetra ($v = n$), la fórmula no cambia y la cota es inmediata. Si penetra, se debe acotar $\max(n, \text{max_var_form}(\text{substForm } v\ u\ f))$. Por hipótesis de inducción, $\text{max_var_form}(\text{substForm } v\ u\ f) \le \max(\text{max_var_form}(f), \text{max_var_term}(u))$. El máximo con n está acotado por $\max(\max(n, \text{max_var_form}(f)), \text{max_var_term}(u))$, que es exactamente la cota requerida.

Significado

Estos lemas son los análogos para variables de los lemas `maxHenkinTerm_substTerm` y `maxHenkinForm_substForm` vistos anteriormente para constantes de Henkin. Establecen que la operación de sustitución es "controlada" respecto a la complejidad de las variables: el resultado no puede contener variables con índices mayores que el máximo de los ya presentes en la fórmula original y en el término de sustitución.

Esta propiedad de monotonía es esencial para la construcción de Henkin-Lindenbaum, donde se necesita garantizar que las nuevas variables introducidas (elegidas como mayores que ciertas cotas) no aparecen en las fórmulas transformadas. También es utilizada en las demostraciones de existencia de variables frescas (`exists_fresh_var_of_deriv`), donde se calcula una cota superior para todas las variables en una derivación y se elige una variable mayor como fresca. La preservación de las cotas bajo sustituciones asegura que esta estrategia es estable: las operaciones sintácticas no introducen variables con índices arbitrariamente grandes que pudieran violar las condiciones de frescura.

La utilización recurrente de `max_le_max_of_le` refleja la estructura de retículo de las cotas: las operaciones sintácticas son monótonas respecto al orden de los máximos, lo que permite un manejo composicional y algebraico de la frescura. Esta es una de las claves que hacen viable la formalización completa de la aritmetización de la sintaxis en un asistente de pruebas como Lean.He completado el comentario pedagógico de todos los fragmentos de código que me has proporcionado. El desarrollo abarca desde

los fundamentos sintácticos del sistema SPA hasta la construcción del modelo canónico y la demostración del Teorema de Verdad, cubriendo:

- Estructuras sintácticas y codificación (HenkinConst, NatS, ArithTerm, ArithForm, Code)

- Serialización, parseo y numerabilidad

- Predicados sintácticos y frescura (NotOccur, NotFreeForm, NoBoundVar)

- Sistema deductivo SPA_Deduction_Internal

- Predicados sobre derivaciones (FreshForDerivation, MaxVarDeriv)

- Álgebra de sustituciones (conmutación, cancelación, preservación)

- Alpha-conversión y cambio de variable ligada

- Construcción de Henkin (GammaWitnessesList, GammaSeq, GammaLimit)

- Consistencia y constantes de Henkin

- Sustitución y eliminación de constantes de Henkin

- Soporte finito de derivaciones (compacidad)

- Construcción de Lindenbaum (LindenbaumSeq, LindenbaumLimit)

- Teorías maximales y propiedad de Henkin

- Términos cerrados y modelo canónico

- Interpretaciones de funciones y predicados en el modelo

- Teorema de Verdad (Truth Lemma)

Si necesitas alguna aclaración adicional o quieres que comente algún fragmento específico con más detalle, no dudes en pedírmelo.

```
--
================================================================
-- FASE 15: SOPORTE MATEMÁTICO, SENTENCIAS Y CONSISTENCIA
--
================================================================

lemma max_var_term_substTerm {s : Nat} (t : ArithTerm s) (v : Nat) (u : ArithTerm s) :
    max_var_term (substTerm v u t) ≤ Nat.max (max_var_term t) (max_var_term u) := by
  induction t with
  | var n =>
    unfold substTerm
    split_ifs with _h_vn
    · exact Nat.le_max_right _ _
    · exact Nat.le_max_left _ _
  | const c' =>
    unfold substTerm; unfold max_var_term
    exact Nat.zero_le _
  | henkin c' =>
    unfold substTerm; unfold max_var_term
    exact Nat.zero_le _
```

```
  | zero =>
    unfold substTerm; unfold max_var_term
    exact Nat.zero_le _
  | succ t' ih =>
    exact ih
  | add t1 t2 ih1 ih2 =>
    exact max_le_max_of_le ih1 ih2
  | mul t1 t2 ih1 ih2 =>
    exact max_le_max_of_le ih1 ih2
  | exp t1 t2 ih1 ih2 =>
    exact max_le_max_of_le ih1 ih2
  | pred t' ih =>
    exact ih

lemma max_var_form_substForm {s : Nat} (φ : ArithForm s) (v : Nat) (u : ArithTerm s)
:
    max_var_form (substForm v u φ) ≤ Nat.max (max_var_form φ) (max_var_term u) :=
by
  induction φ with
  | bottom =>
    unfold substForm; unfold max_var_form
    exact Nat.zero_le _
  | eq t1 t2 =>
    exact max_le_max_of_le (max_var_term_substTerm t1 v u)
(max_var_term_substTerm t2 v u)
  | impl f1 f2 ih1 ih2 =>
    exact max_le_max_of_le ih1 ih2
  | neg f ih =>
    exact ih
```

```
  | forall_form n f ih =>
    unfold substForm
    split_ifs with _h_vn
    · exact Nat.le_max_left _ _
    · dsimp [max_var_form]
      apply Nat.max_le.mpr
      constructor
      · have : n ≤ Nat.max n (max_var_form f) := Nat.le_max_left n (max_var_form f)
        exact Nat.le_trans this (Nat.le_max_left (Nat.max n (max_var_form f)) (max_var_term u))
      · apply Nat.le_trans ih
        apply Nat.max_le.mpr
        constructor
        · have : max_var_form f ≤ Nat.max n (max_var_form f) := Nat.le_max_right n (max_var_form f)
          exact Nat.le_trans this (Nat.le_max_left (Nat.max n (max_var_form f)) (max_var_term u))
        · exact Nat.le_max_right (Nat.max n (max_var_form f)) (max_var_term u)
  | provable_op heq t =>
    exact max_var_term_substTerm t v u
  | true_op heq t =>
    exact max_var_term_substTerm t v u
  | list_nth_eq t1 t2 t3 =>
    exact max_le_max_of_le (max_var_term_substTerm t1 v u)
      (max_le_max_of_le (max_var_term_substTerm t2 v u) (max_var_term_substTerm t3 v u))
```

Lemas de preservación de no ligadura y no libertad bajo sustituciones

Se demuestran tres lemas que completan la caracterización del comportamiento de las propiedades de frescura bajo sustituciones. Estos resultados son utilizados en los lemas de cambio de variable ligada y en la construcción de testigos de Henkin.

Lema `NoBoundVar_substForm_var`: Preservación de no ligadura bajo auto-sustitución

Establece que si una variable w no está ligada en una fórmula φ, entonces w sigue sin estar ligada en φ tras sustituir cualquier variable v por $\text{var } w$:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \text{NoBoundVar } w\ \varphi \implies \text{NoBoundVar } w\ (\text{substForm } v\ (\text{var } w)\ \varphi) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos base y sin cuantificadores:** `NoBoundVar` es siempre verdadero.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** La hipótesis `NoBoundVar w (forall_form n f)` da $w \neq n$ y `NoBoundVar w f`. Si $v = n$, la sustitución no penetra y la propiedad se preserva. Si $v \neq n$, la sustitución atraviesa el cuantificador. Se debe verificar que w no está ligada en $\forall n\ (\text{substForm } v\ (\text{var } w)\ f)$. La condición $w \neq n$ se mantiene, y `NoBoundVar w` en el cuerpo se sigue de la hipótesis de inducción.

Significado

Este lema garantiza que la operación de sustituir una variable por w no introduce nuevas ligaduras de w. Es importante para los lemas de alpha-conversión, donde se renombra una variable ligada a una fresca: la nueva variable no debe estar ya ligada en la fórmula. La preservación de `NoBoundVar` bajo la operación inversa asegura que la alpha-conversión es reversible.

Lema `NotOccur_substTerm_self`: No ocurrencia tras auto-sustitución

Establece que si una variable v no ocurre en un término t, entonces v tampoco ocurre en el resultado de sustituir v por t en cualquier término t':

$$ \forall t' \in \text{ArithTerm } s,\ \forall v \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \text{NotOccur } v\ t \implies \text{NotOccur } v\ (\text{substTerm } v\ t\ t') $$

Estructura de la demostración

Por inducción sobre t':

- **`var n`:** Si $n = v$, se reemplaza por t, y v no ocurre en t por hipótesis. Si $n \neq v$, se preserva $\text{var } n$, y $v \neq n$ implica que v no ocurre.

- **Casos base:** Triviales.

- **Casos recursivos:** Se invierten o propagan las hipótesis.

Significado

Al sustituir v por t, todas las ocurrencias de v son reemplazadas por t. Si v no ocurría en t, entonces v no puede aparecer en el resultado. Este lema es la base para `NotFreeForm_substForm_self`.

Lema `NotFreeForm_substForm_self`: Variable no libre tras auto-sustitución

Establece que si t es un término cerrado, entonces al sustituir v por t en cualquier fórmula ψ, la variable v deja de ser libre en el resultado:

$$ \forall \psi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \text{isClosed } t = \text{true} \implies \text{NotFreeForm } v\ (\text{substForm } v\ t\ \psi) $$

Estructura de la demostración

Por inducción sobre ψ, utilizando `NotOccur_substTerm_self` para los casos con términos (gracias a que t es cerrado, v no ocurre en t por `NotOccur_of_isClosed`). El caso `forall_form n f` distingue si $v = n$ (la sustitución no penetra, y v está ligada por el cuantificador, por lo que no es libre) o $v \neq n$ (la sustitución se propaga, y por hipótesis de inducción v no es libre en el cuerpo).

Significado

Este lema captura la propiedad fundamental de la operación de grounding (o instanciación): al reemplazar una variable libre por un término cerrado, la variable desaparece de la fórmula. Es la razón por la que la operación `substAll` (que sustituye todas las variables por términos cerrados) produce fórmulas cerradas, y es esencial

para la definición de satisfacción en el modelo canónico, donde las fórmulas con variables libres se evalúan cerrando todas las variables mediante el entorno de asignación. La condición de que t sea cerrado es crucial: si t contuviera a v, la sustitución no eliminaría v, sino que la reintroduciría.

Estos tres lemas, junto con los anteriores sobre preservación de `NotFreeForm` y `NoBoundVar` bajo sustituciones, completan el cuadro completo de cómo las propiedades de frescura se comportan bajo las operaciones sintácticas fundamentales, proporcionando las garantías necesarias para todas las construcciones del sistema SPA.

```
lemma NoBoundVar_substForm_var {s : Nat} (φ : ArithForm s) (v w : Nat) (h : NoBoundVar w φ) :
  NoBoundVar w (substForm v (ArithTerm.var w) φ) := by
 induction φ with
 | bottom => trivial
 | eq _ _ => trivial
 | impl f1 f2 ih1 ih2 =>
  dsimp [substForm, NoBoundVar] at h ⊢
  exact ⟨ih1 h.1, ih2 h.2⟩
 | neg f ih =>
  dsimp [substForm, NoBoundVar] at h ⊢
  exact ih h
 | forall_form n f ih =>
  dsimp [substForm, NoBoundVar] at h ⊢
  by_cases h_vn : v = n
```

```
    · rw [if_pos h_vn]
      exact h
    · rw [if_neg h_vn]
      dsimp [NoBoundVar]
      exact ⟨h.1, ih h.2⟩
  | provable_op _ _ => trivial
  | true_op _ _ => trivial
  | list_nth_eq _ _ _ => trivial

lemma NotOccur_substTerm_self {s : Nat} (t' : ArithTerm s) (v : Nat) (t : ArithTerm s)
(h_t : NotOccur v t) :
    NotOccur v (substTerm v t t') := by
  induction t' with
  | var n =>
    unfold substTerm
    split_ifs with h_vn
    · exact h_t
    · exact NotOccur.var n h_vn
  | const c => exact NotOccur.const c
  | henkin c => exact NotOccur.henkin c
  | zero => exact NotOccur.zero
  | succ t'' ih => unfold substTerm; exact NotOccur.succ _ ih
  | add t1 t2 ih1 ih2 => unfold substTerm; exact NotOccur.add _ _ ih1 ih2
  | mul t1 t2 ih1 ih2 => unfold substTerm; exact NotOccur.mul _ _ ih1 ih2
  | exp t1 t2 ih1 ih2 => unfold substTerm; exact NotOccur.exp _ _ ih1 ih2
  | pred t'' ih => unfold substTerm; exact NotOccur.pred _ ih

lemma NotFreeForm_substForm_self {s : Nat} (ψ : ArithForm s) (v : Nat) (t :
ArithTerm s) (h_closed : isClosed t = true) :
```

```
  NotFreeForm v (substForm v t ψ) := by
have h_not : NotOccur v t := NotOccur_of_isClosed t h_closed v
induction ψ with
| bottom =>
  dsimp [substForm, NotFreeForm]
| eq t1 t2 =>
  dsimp [substForm, NotFreeForm]
  exact ⟨NotOccur_substTerm_self t1 v t h_not, NotOccur_substTerm_self t2 v t h_not⟩
| impl f1 f2 ih1 ih2 =>
  dsimp [substForm, NotFreeForm]
  exact ⟨ih1, ih2⟩
| neg f ih =>
  dsimp [substForm, NotFreeForm]
  exact ih
| forall_form n f ih =>
  dsimp [substForm]
  by_cases h_vn : v = n
  · rw [if_pos h_vn]
    dsimp [NotFreeForm]
    rw [if_pos h_vn]
    trivial
  · rw [if_neg h_vn]
    dsimp [NotFreeForm]
    rw [if_neg h_vn]
    exact ih
| provable_op heq t' =>
  dsimp [substForm, NotFreeForm]
  exact NotOccur_substTerm_self t' v t h_not
```

```
  | true_op heq t' =>
    dsimp [substForm, NotFreeForm]
    exact NotOccur_substTerm_self t' v t h_not
  | list_nth_eq t1 t2 t3 =>
    dsimp [substForm, NotFreeForm]
    exact ⟨NotOccur_substTerm_self t1 v t h_not, NotOccur_substTerm_self t2 v t h_not,
NotOccur_substTerm_self t3 v t h_not⟩
```

Lemas auxiliares para sentencias y frescura de variables

Se demuestran cuatro lemas que abordan propiedades específicas sobre fórmulas existenciales, generación de variables frescas, e invarianza de términos bajo sustituciones que no afectan a sus variables.

Lema `NotFreeForm_existential`: Variable ligada en una fórmula existencial

Establece que en una fórmula existencial $\exists v\ \psi$ (representada como $\neg \forall v\ \neg \psi$), la variable v no es libre (está ligada):

$$ \forall v \in \mathbb{N},\ \forall \psi \in \text{ArithForm } 0,\ \text{NotFreeForm } v\ (\neg \forall v\ \neg \psi) $$

La demostración es inmediata por la definición de `NotFreeForm`: para el cuantificador `forall_form`, cuando la variable consultada coincide con la variable ligada, la condición `if_pos rfl` se evalúa a `True`. La doble negación externa no afecta a la libertad de v.

Significado

Este lema confirma que la representación de la cuantificación existencial como $\neg \forall \neg$ respeta la noción estándar de variable ligada. Es utilizado en la propiedad de Henkin y en la construcción de testigos, donde se manipulan fórmulas de la forma $\neg \forall v\ \neg \psi$ y se necesita saber que v no es libre en ellas.

Definición `freshVar` y lema `freshVar_gt`: Generación de variable fresca

`freshVar L`: Define una variable fresca respecto a una lista de fórmulas L, tomando el máximo de variables más uno:

$$ \text{freshVar}(L) := \text{max_var_list}(L) + 1 $$

`freshVar_gt`: Establece que para cualquier fórmula φ en la lista L, el máximo de variables de φ es estrictamente menor que `freshVar L`:

$$ \forall \varphi \in L,\ \text{max_var_form}(\varphi) < \text{freshVar}(L) $$

La demostración utiliza `max_var_list_ge` para obtener $\text{max_var_form}(\varphi) \le \text{max_var_list}(L)$, y luego $\text{max_var_list}(L) < \text{max_var_list}(L) + 1 = \text{freshVar}(L)$.

Significado

Esta es la implementación concreta de la estrategia de "frescura por cota superior" para variables. `freshVar` proporciona una variable que garantizadamente no aparece en ninguna fórmula de la lista L, ya que su índice es estrictamente mayor que cualquier variable en L. Es utilizada en la construcción de testigos de Henkin y en los lemas de cambio de variable ligada.

Lema `substAllTerm_sentencia_general`: Invarianza bajo sustituciones que no afectan

Establece una condición suficiente para que `substAllTerm` no modifique un término: si para cada variable v, o bien v no ocurre en t, o bien el entorno mapea v a $\text{var } v$ (es decir, no la modifica), entonces la sustitución global deja t inalterado:

$$ \forall t \in \text{ArithTerm } s,\ \forall \text{env} : \mathbb{N} \to \text{ArithTerm } s, $$

$$ (\forall v,\ \text{NotOccur } v\ t \lor \text{env } v = \text{var } v) \implies \text{substAllTerm } t\ \text{env} = t $$

Estructura de la demostración

Por inducción sobre t:

- **`var n`:** La hipótesis para $v = n$ da dos casos: si n no ocurre en `var n`, es una contradicción (pues `var n` contiene a n). Por tanto, debe darse $\text{env } n = \text{var } n$, y la sustitución preserva la variable.

- **Casos base (`const`, `henkin`, `zero`):** `substAllTerm` no modifica estos constructores.

- **Casos recursivos:** Se especializa la hipótesis para cada subtérmino. Por ejemplo, para `succ t'`, si v no ocurre en `succ t'`, entonces v no ocurre en t' (por inversión de `NotOccur`). Similarmente para `add`, `mul`, `exp`, `pred`. Esto permite aplicar las hipótesis de inducción a los subtérminos.

Significado

Este lema captura la idea de que `substAllTerm` solo modifica las variables que el entorno cambia efectivamente. Si el entorno es la identidad sobre todas las variables que ocurren en t, la sustitución global es la identidad sobre t. Este resultado es útil en el Teorema de Verdad: cuando se evalúa una fórmula en el modelo canónico bajo una asignación, `substAll` se aplica a los términos. Si la asignación no modifica ciertas variables (por ejemplo, variables ligadas que se mapean a sí mismas), los términos que solo contienen esas variables no se alteran.

La condición disyuntiva permite un tratamiento uniforme: para cada variable, o bien no está en el término (y no importa qué haga el entorno), o bien el entorno la preserva. Esto es exactamente lo que ocurre con las variables ligadas en la interpretación de cuantificadores, donde el entorno modificado mapea la variable ligada a $\text{var } n$ (se preserva a sí misma), mientras que las demás variables pueden ser modificadas. La hipótesis de inducción sobre la estructura del término permite propagar esta propiedad desde las hojas hasta la raíz.

lemma NotFreeForm_existential (v : Nat) (ψ : ArithForm 0) :

```
  NotFreeForm v (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) := by
 unfold NotFreeForm
 unfold NotFreeForm
 rw [if_pos rfl]
 trivial


def freshVar (L : List (ArithForm 0)) : Nat :=
 max_var_list L + 1


lemma freshVar_gt (L : List (ArithForm 0)) (φ : ArithForm 0) (h : φ ∈ L) :
max_var_form φ < freshVar L := by
 unfold freshVar
 have h_max : max_var_form φ ≤ max_var_list L := max_var_list_ge L φ h
 omega


lemma substAllTerm_sentencia_general {s : Nat} (t : ArithTerm s) (env : Nat →
ArithTerm s)
   (h_id : ∀ v, NotOccur v t ∨ env v = ArithTerm.var v) :
   substAllTerm t env = t := by
 induction t with
 | var n =>
   dsimp [substAllTerm]
   have h := h_id n
   cases h with
   | inl h_not =>
     cases h_not with | var _ h_ne => contradiction
   | inr h_eq => exact h_eq
 | const c => rfl
 | henkin c => rfl
```

```
| zero => rfl
| succ t' ih =>
  dsimp [substAllTerm]
  have h_not : ∀ v, NotOccur v t' ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not =>
      cases h_not with | succ _ h' => left; exact h'
    | inr h_eq => right; exact h_eq
  rw [ih h_not]
| add t1 t2 ih1 ih2 =>
  dsimp [substAllTerm]
  have h1 : ∀ v, NotOccur v t1 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not =>
      cases h_not with | add _ _ h1' _ => left; exact h1'
    | inr h_eq => right; exact h_eq
  have h2 : ∀ v, NotOccur v t2 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not =>
      cases h_not with | add _ _ _ h2' => left; exact h2'
    | inr h_eq => right; exact h_eq
  rw [ih1 h1, ih2 h2]
```

```
| mul t1 t2 ih1 ih2 =>
  dsimp [substAllTerm]
  have h1 : ∀ v, NotOccur v t1 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not =>
      cases h_not with | mul _ _ h1' _ => left; exact h1'
    | inr h_eq => right; exact h_eq
  have h2 : ∀ v, NotOccur v t2 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not =>
      cases h_not with | mul _ _ _ h2' => left; exact h2'
    | inr h_eq => right; exact h_eq
  rw [ih1 h1, ih2 h2]
| exp t1 t2 ih1 ih2 =>
  dsimp [substAllTerm]
  have h1 : ∀ v, NotOccur v t1 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not =>
      cases h_not with | exp _ _ h1' _ => left; exact h1'
    | inr h_eq => right; exact h_eq
  have h2 : ∀ v, NotOccur v t2 ∨ env v = ArithTerm.var v := by
    intro v
```

```
    have h := h_id v
    cases h with
    | inl h_not =>
      cases h_not with | exp _ _ _ h2' => left; exact h2'
    | inr h_eq => right; exact h_eq
  rw [ih1 h1, ih2 h2]
| pred t' ih =>
  dsimp [substAllTerm]
  have h_not : ∀ v, NotOccur v t' ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not =>
      cases h_not with | pred _ h' => left; exact h'
    | inr h_eq => right; exact h_eq
  rw [ih h_not]
```

Lema `substAll_raw_sentencia_general`: Invarianza de fórmulas bajo sustituciones que no afectan

Se demuestra la extensión del lema `substAllTerm_sentencia_general` al nivel de fórmulas. Este resultado establece que `substAll_raw` no modifica una fórmula cuando el entorno de sustitución no altera ninguna variable que ocurra en ella.

Enunciado formal

$$ \forall \varphi \in \text{ArithForm } s,\ \forall \text{env} : \mathbb{N} \to \text{ArithTerm } s, $$

$$ (\forall v,\ \text{NotOccurForm } v\ \varphi \lor \text{env } v = \text{var } v) \implies \text{substAll_raw } \varphi\ \text{env} = \varphi $$

Estructura de la demostración

Por inducción sobre φ:

- **`bottom`:** Trivial, `substAll_raw` no modifica `bottom`.

- **`eq t1 t2`:** De la hipótesis, para cada v, o bien `NotOccurForm v (eq t1 t2)` (lo que implica `NotOccur v t1` y `NotOccur v t2`), o bien `env v = var v`. Se extraen las hipótesis para t_1 y t_2 por separado y se aplica `substAllTerm_sentencia_general` a cada uno.

- **`impl f1 f2`:** Similar, descomponiendo `NotOccurForm` en sus dos componentes y aplicando las hipótesis de inducción a f_1 y f_2.

- **`neg f`:** `NotOccurForm v (neg f)` es equivalente a `NotOccurForm v f`, por lo que la hipótesis se transfiere directamente y se aplica la inducción.

- **`forall_form n f`:** Este es el caso más interesante. Se debe demostrar que `substAll_raw` con el entorno modificado (que mapea n a $\text{var } n$) deja f inalterado. Para aplicar la hipótesis de inducción, se necesita verificar la condición para todas las variables v con el nuevo entorno. Dado v:

  - Si $v = n$: El nuevo entorno mapea n a $\text{var } n$, por lo que la segunda alternativa (igualdad con $\text{var } v$) se cumple.

  - Si $v \neq n$: El nuevo entorno coincide con `env v`. De la hipótesis original para $\forall n\ f$, se tiene `NotOccurForm v (forall_form n f)` o `env v = var v`. En el primer caso, `NotOccurForm v (forall_form n f)` implica $v \neq n$ y `NotOccurForm v f`, por lo que la primera alternativa se cumple para f. En el segundo caso, la segunda alternativa se cumple.

- **`provable_op h t`, `true_op h t`:** Se aplica `substAllTerm_sentencia_general` al término t, ya que `NotOccurForm v` para estos constructores es equivalente a `NotOccur v t`.

- **`list_nth_eq t1 t2 t3`:** Similar a `eq`, extrayendo las condiciones para cada uno de los tres términos.

Significado

`substAll_raw_sentencia_general` es un resultado fundamental para el Teorema de Verdad y la definición de satisfacción en el modelo canónico. Establece que la operación `substAll_raw` (que sustituye todas las variables según un entorno) es la identidad sobre las fórmulas siempre que el entorno no modifique ninguna variable que ocurra en la fórmula. Esto tiene dos consecuencias importantes:

1. **Fórmulas cerradas:** Si φ es una fórmula cerrada (ninguna variable ocurre en ella), entonces para cualquier entorno, `substAll_raw φ env = φ`. Esto

significa que la verdad de una fórmula cerrada no depende de la asignación, lo cual es exactamente lo que se espera semánticamente.

2. **Variables ligadas:** En la interpretación del cuantificador universal, el entorno se modifica para que la variable ligada n se mapee a sí misma ($\text{var } n$). La condición del lema se satisface para este entorno modificado porque para $v = n$ se cumple la igualdad, y para $v \neq n$ la hipótesis original sobre `NotOccurForm` proporciona la otra alternativa. Esto justifica que la interpretación de $\forall n\ f$ bajo el entorno modificado dependa de f bajo ese mismo entorno.

El lema es, en esencia, la formalización de que `substAll_raw` es una operación de "cierre" que solo afecta a las variables libres, dejando inalteradas las partes de la fórmula donde las variables no ocurren o están ligadas. Esta propiedad es la que permite que la definición de `satisfies` en el modelo canónico sea correcta y coincida con la noción semántica estándar de satisfacción de Tarski.

```
lemma substAll_raw_sentencia_general {s : Nat} (φ : ArithForm s) (env : Nat → ArithTerm s)
  (h_id : ∀ v, NotOccurForm v φ ∨ env v = ArithTerm.var v) :
  substAll_raw φ env = φ := by
 induction φ generalizing env
 · -- bottom
   rfl
 · -- eq
```

```
  rename_i t1 t2
  dsimp [substAll_raw]
  have h1 : ∀ v, NotOccur v t1 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    unfold NotOccurForm at h
    cases h with
    | inl h_not => rcases h_not with ⟨h_not1, h_not2⟩; left; exact h_not1
    | inr h_eq => right; exact h_eq
  have h2 : ∀ v, NotOccur v t2 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    unfold NotOccurForm at h
    cases h with
    | inl h_not => rcases h_not with ⟨h_not1, h_not2⟩; left; exact h_not2
    | inr h_eq => right; exact h_eq
  rw [substAllTerm_sentencia_general t1 env h1, substAllTerm_sentencia_general t2
env h2]
 · -- impl
  rename_i f1 f2 ih1 ih2
  dsimp [substAll_raw]
  have h1 : ∀ v, NotOccurForm v f1 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    unfold NotOccurForm at h
    cases h with
    | inl h_not => rcases h_not with ⟨h_not1, h_not2⟩; left; exact h_not1
    | inr h_eq => right; exact h_eq
```

```
  have h2 : ∀ v, NotOccurForm v f2 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    unfold NotOccurForm at h
    cases h with
    | inl h_not => rcases h_not with ⟨h_not1, h_not2⟩; left; exact h_not2
    | inr h_eq => right; exact h_eq
  rw [ih1 env h1, ih2 env h2]
· -- neg
  rename_i f ih
  dsimp [substAll_raw]
  have h1 : ∀ v, NotOccurForm v f ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not => left; exact h_not
    | inr h_eq => right; exact h_eq
  rw [ih env h1]
· -- forall_form
  rename_i n f ih
  dsimp [substAll_raw]
  have h_not : ∀ v, NotOccurForm v f ∨ (if v = n then ArithTerm.var n else env v) =
ArithTerm.var v := by
    intro v
    by_cases h_vn : v = n
    · subst h_vn
      rw [if_pos rfl]
      right; rfl
```

```
    · rw [if_neg h_vn]
      have h := h_id v
      cases h with
      | inl h_not_forall =>
        left; exact h_not_forall.2
      | inr h_eq =>
        right; exact h_eq
  rw [ih _ h_not]
· -- provable_op
  rename_i s_l h_eq t
  dsimp [substAll_raw]
  have h1 : ∀ v, NotOccur v t ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not => left; exact h_not
    | inr h_eq => right; exact h_eq
  rw [substAllTerm_sentencia_general t env h1]
· -- true_op
  rename_i s_l h_eq t
  dsimp [substAll_raw]
  have h1 : ∀ v, NotOccur v t ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    cases h with
    | inl h_not => left; exact h_not
    | inr h_eq => right; exact h_eq
  rw [substAllTerm_sentencia_general t env h1]
```

```
· -- list_nth_eq
  rename_i t1 t2 t3
  dsimp [substAll_raw]
  have h1 : ∀ v, NotOccur v t1 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    unfold NotOccurForm at h
    cases h with
    | inl h_not => rcases h_not with ⟨h_not1, h_not2, h_not3⟩; left; exact h_not1
    | inr h_eq => right; exact h_eq
  have h2 : ∀ v, NotOccur v t2 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    unfold NotOccurForm at h
    cases h with
    | inl h_not => rcases h_not with ⟨h_not1, h_not2, h_not3⟩; left; exact h_not2
    | inr h_eq => right; exact h_eq
  have h3 : ∀ v, NotOccur v t3 ∨ env v = ArithTerm.var v := by
    intro v
    have h := h_id v
    unfold NotOccurForm at h
    cases h with
    | inl h_not => rcases h_not with ⟨h_not1, h_not2, h_not3⟩; left; exact h_not3
    | inr h_eq => right; exact h_eq
  rw [substAllTerm_sentencia_general t1 env h1, substAllTerm_sentencia_general t2
env h2, substAllTerm_sentencia_general t3 env h3]
```

Lemas sobre sentencias, constantes de Henkin y consistencia

Se demuestran tres lemas que relacionan los conceptos de fórmula cerrada (sentencia), constantes de Henkin, y consistencia de teorías.

Lema `substAll_sentencia`: Invarianza de sentencias bajo sustitución global

Establece que si una fórmula φ es una sentencia (no contiene ninguna variable en absoluto), entonces `substAll` no la modifica, independientemente del entorno:

$$ \forall \varphi \in \text{ArithForm } s,\ (\forall v,\ \text{NotOccurForm } v\ \varphi) \implies \forall \text{env} : \mathbb{N} \to \text{TermClosed } s,\ \text{substAll } \varphi\ \text{env} = \varphi $$

La demostración es una aplicación directa de `substAll_raw_sentencia_general`: la hipótesis de que φ no contiene ninguna variable proporciona exactamente la primera alternativa de la condición requerida.

Significado

Este lema confirma que las sentencias (fórmulas sin variables) son invariantes bajo la operación de sustitución global. En el contexto del Teorema de Verdad, esto implica que la satisfacción de una sentencia no depende de la asignación de variables, lo cual es semánticamente esperado: una fórmula sin variables libres tiene un valor de verdad absoluto en el modelo. La propiedad de Henkin trata precisamente con

sentencias (o fórmulas cerradas), y este lema garantiza que la operación `substAll` utilizada en la definición de `satisfies` no altera las sentencias.

Lema `maxHenkinForm_neg`: El máximo de constantes de Henkin es invariante bajo negación

Establece la igualdad obvia por definición:

$$ \text{maxHenkinForm}(\neg \varphi) = \text{maxHenkinForm}(\varphi) $$

La demostración es `rfl` porque `maxHenkinForm` para `neg` está definido como el máximo de la subfórmula.

Lema `no_deduccion_implica_consistencia_neg`: Consistencia de la negación

Establece una conexión entre la no demostrabilidad y la consistencia: si una fórmula φ no es demostrable a partir del conjunto vacío, entonces el conjunto que contiene únicamente a $\neg\varphi$ es consistente:

$$ \forall \varphi \in \text{ArithForm } 0,\ \neg(\emptyset \vdash \varphi) \implies \text{Consistent}(\{\neg\varphi\}) $$

Estructura de la demostración

Se asume que $\{\neg\varphi\}$ es inconsistente, es decir, existe una derivación de $\bot$ a partir de $\{\neg\varphi\}$.

1. Por `weaken`, esta derivación se extiende al contexto $\emptyset \cup \{\neg\varphi\}$.

2. Aplicando `neg_intro` a esta derivación, se obtiene $\emptyset \vdash \neg\neg\varphi$.

3. Por `double_neg`, se obtiene $\emptyset \vdash \varphi$.

4. Esto contradice la hipótesis de que φ no es demostrable.

Significado

Este lema es una versión del principio de que si una fórmula no es un teorema lógico, entonces su negación es consistente (no conduce a una contradicción). Es utilizado en la construcción de Lindenbaum: cuando se procesa una fórmula φ y esta no puede añadirse consistentemente al conjunto acumulado, el lema `consistent_insert_neg` (demostrado anteriormente) garantiza que añadir $\neg\varphi$ preserva la consistencia. El presente lema proporciona la justificación de que, en el caso base (cuando el conjunto acumulado es vacío o solo contiene negaciones), la negación de una fórmula no demostrable es consistente.

La restricción al estrato $s = 0$ es incidental; el resultado vale para cualquier estrato, pero se enuncia para 0 porque es el caso utilizado en la construcción de Lindenbaum para el nivel base. La demostración utiliza únicamente reglas del sistema deductivo (`weaken`, `neg_intro`, `double_neg`) y la definición de `Consistent`, mostrando cómo la maquinaria deductiva permite transferir la no demostrabilidad a una propiedad de consistencia.

```
lemma substAll_sentencia {s : Nat} {φ : ArithForm s} (h_sentencia : ∀ v, NotOccurForm v φ) (env : Nat → TermClosed s) :
  substAll φ env = φ := by
 dsimp [substAll]
 apply substAll_raw_sentencia_general
 intro v
 left
 exact h_sentencia v

lemma maxHenkinForm_neg (φ : ArithForm 0) : maxHenkinForm (ArithForm.neg φ) = maxHenkinForm φ := rfl

lemma no_deduccion_implica_consistencia_neg {φ : ArithForm 0} (h_no_ded : ¬ SPA_Deduction_Internal 0 ∅ φ) :
  Consistent ({ArithForm.neg φ} : Set (ArithForm 0)) := by
 intro h_bot
 have h_bot_insert : SPA_Deduction_Internal 0 (insert (ArithForm.neg φ) (∅ : Set (ArithForm 0))) ArithForm.bottom := by
  apply SPA_Deduction_Internal.weaken ({ArithForm.neg φ} : Set (ArithForm 0)) _ _ h_bot
  intro x hx
  rcases hx with rfl
  exact Or.inl rfl
 have h_nnphi : SPA_Deduction_Internal 0 ∅ (ArithForm.neg (ArithForm.neg φ)) :=
  SPA_Deduction_Internal.neg_intro (∅ : Set (ArithForm 0)) (ArithForm.neg φ) h_bot_insert
 have h_phi : SPA_Deduction_Internal 0 ∅ φ :=
  SPA_Deduction_Internal.double_neg (∅ : Set (ArithForm 0)) φ h_nnphi
```

exact h_no_ded h_phi

Lemas de acotación para la decodificación y frescura condicional

Se demuestran dos lemas que proporcionan control sobre las cotas de constantes de Henkin en la enumeración de fórmulas y sobre la libertad de variables en conjuntos con máximo de variables acotado.

Lema `maxHenkinForm_decodeExists_le_GammaWitnessesList_succ`: Cota para la fórmula decodificada

Establece que el máximo índice de constante de Henkin en la fórmula decodificada en el paso m está acotado por el máximo de la lista de testigos acumulada hasta el paso $m+1$:

$$ \forall m \in \mathbb{N},\ \text{maxHenkinForm}(\text{decodeExists } m) \le \text{maxHenkinList}(\text{GammaWitnessesList } (m+1)) $$

Estructura de la demostración

Se analiza la definición de `GammaWitnessesList` en el paso $m+1$, distinguiendo dos casos según si `getExistentialComponents` devuelve `none` o `some (v, ψ)`:

- **Caso `none`:** La lista no se modifica, y `decodeExists m` es `defaultExists 0`, que tiene `maxHenkinForm = 0`. La cota se cumple trivialmente.

- **Caso `some (v, ψ)`:** Se añade el axioma de testigo a la lista. Se demuestra que `maxHenkinForm(decodeExists m) = maxHenkinForm(freshEx)`, donde `freshEx` es la fórmula existencial construida. Esta igualdad se debe a que `decodeExists m` es exactamente $\neg \forall v\ \neg \psi$, y `freshEx` es $\neg \forall w\ \neg (\psi[v := \text{var } w])$. Por el lema `maxHenkinForm_substForm_var` (que establece que la sustitución de una variable por otra no modifica el máximo de constantes de Henkin), ambas fórmulas tienen el mismo máximo. El máximo de `freshEx` está incluido en el máximo de la lista extendida, por lo que la cota se cumple.

Lema `NotFreeInSet_of_max_var_form_zero`: Frescura para variables positivas en conjuntos acotados

Establece que si todas las fórmulas de un conjunto S tienen máximo de variables igual a 0 (es decir, no contienen variables en absoluto), entonces cualquier variable $w > 0$ no es libre en ninguna fórmula de S:

$$ \forall S \subseteq \text{ArithForm } 0,\ (\forall \psi \in S,\ \text{max_var_form}(\psi) = 0) \implies \forall w > 0,\ \text{NotFreeInSet } w\ S $$

Estructura de la demostración

Para cada $\psi \in S$:

1. De la hipótesis, $\text{max_var_form}(\psi) = 0$.

2. Como $w > 0$, se tiene $w > \text{max_var_form}(\psi)$.

3. Por `NotOccurForm_of_gt_max_var`, w no ocurre en ψ.

4. Por `NotFreeForm_of_NotOccurForm`, w no es libre en ψ.

Significado

Este lema proporciona una condición suficiente simple para que una variable sea fresca respecto a un conjunto: si el conjunto solo contiene fórmulas sin variables (como axiomas lógicos o fórmulas cerradas), entonces cualquier variable positiva es automáticamente no libre en el conjunto. Es utilizado en las etapas iniciales de la construcción de Henkin, donde el conjunto de partida Γ a menudo satisface esta condición (por ejemplo, si Γ contiene solo sentencias o axiomas sin variables). La condición $\text{max_var_form}(\psi) = 0$ es una forma fuerte de decir que ψ no contiene variables, lo cual es típico de los axiomas de Peano y otras fórmulas básicas una vez que sus variables han sido instanciadas.

Ambos lemas son piezas auxiliares para la demostración de que la construcción de Henkin preserva la consistencia: el primero controla las constantes de Henkin introducidas, y el segundo controla las variables libres en el contexto inicial, asegurando que las nuevas variables y constantes elegidas son efectivamente frescas. Juntos, contribuyen a la verificación de que las hipótesis de `consistency_preservation_local` se satisfacen en cada paso de la construcción iterativa.

--
==========

```
-- FASE 16a: SOPORTE DE COTAS DE DECODIFICACIÓN Y GENERALIZACIÓN DE COTAS
--
================================================================================




lemma maxHenkinForm_decodeExists_le_GammaWitnessesList_succ (m : Nat) :
  maxHenkinForm (decodeExists (s := 0) m) ≤ maxHenkinList (GammaWitnessesList (m + 1)) := by
 dsimp [GammaWitnessesList]
 generalize h_dec : getExistentialComponents (s := 0) (decodeExists (s := 0) m) = opt
 cases opt with
 | none =>
  have h_eq := decodeExists_eq_defaultExists_of_getExistentialComponents_none m h_dec
  rw [h_eq]
  change 0 ≤ maxHenkinList (GammaWitnessesList m)
  omega
 | some pair =>
  rcases pair with ⟨v, ψ⟩
  dsimp [maxHenkinList, maxHenkinForm]
  let w := Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) + 1
  let freshEx := ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)))
  let c : HenkinConst := ⟨Nat.max (maxHenkinList (GammaWitnessesList m)) (maxHenkinForm freshEx) + 1⟩
  have h_eq : maxHenkinForm (decodeExists (s := 0) m) = maxHenkinForm freshEx := by
    dsimp [freshEx, maxHenkinForm]
```

```
    rw [maxHenkinForm_substForm_var ψ v]
    rw [decodeExists_eq_of_getExistentialComponents_some m v ψ h_dec]
    rfl
  have h_le1 : maxHenkinForm (decodeExists (s := 0) m) ≤ Nat.max (maxHenkinForm freshEx) (maxHenkinForm (substForm v (ArithTerm.henkin c) ψ)) := by
    rw [h_eq]
    exact Nat.le_max_left _ _
  have h_le2 : Nat.max (maxHenkinForm freshEx) (maxHenkinForm (substForm v (ArithTerm.henkin c) ψ)) ≤
        Nat.max (Nat.max (maxHenkinForm freshEx) (maxHenkinForm (substForm v (ArithTerm.henkin c) ψ))) (maxHenkinList (GammaWitnessesList m)) :=
    Nat.le_max_left _ _
  exact Nat.le_trans h_le1 h_le2

lemma NotFreeInSet_of_max_var_form_zero (S : Set (ArithForm 0)) (hS : ∀ ψ ∈ S, max_var_form ψ = 0) (w : Nat) (hw : w > 0) :
  NotFreeInSet w S := by
 intro ψ hψ
 have h_max : max_var_form ψ = 0 := hS ψ hψ
 have h_not_occ : NotOccurForm w ψ := NotOccurForm_of_gt_max_var ψ w (by omega)
 exact NotFreeForm_of_NotOccurForm w ψ h_not_occ
```

Lemas `max_var_term_eq_zero` y `max_var_form_eq_zero`: Caracterización de sentencias

Se demuestran dos lemas que establecen la equivalencia entre la ausencia total de variables en un término o fórmula y el hecho de que su máximo índice de variable sea cero. Estos resultados proporcionan una caracterización cuantitativa de las sentencias (fórmulas sin variables).

Lema `max_var_term_eq_zero`: Términos sin variables tienen máximo cero

Establece que si ninguna variable ocurre en un término t, entonces el máximo índice de variable en t es 0:

$$ \forall t \in \text{ArithTerm } s,\ (\forall v \in \mathbb{N},\ \text{NotOccur } v\ t) \implies \text{max_var_term}(t) = 0 $$

Estructura de la demostración

Por casos sobre la estructura de t (usando `match`):

- **`var n`:** La hipótesis de que ninguna variable ocurre en `var n` es contradictoria, ya que n ocurre en `var n`. Este caso es imposible.

- **Casos base (`const`, `henkin`, `zero`):** Por definición, `max_var_term` es 0.

- **`succ t'`, `pred t'`:** La hipótesis de que ninguna variable ocurre en el término compuesto implica que ninguna variable ocurre en el subtérmino (por inversión de `NotOccur`). Se aplica la hipótesis de inducción al subtérmino, que da

$\text{max_var_term}(t') = 0$. Como `max_var_term` para estos constructores es igual al del subtérmino, el resultado es 0.

- **`add t1 t2`, `mul t1 t2`, `exp t1 t2`:** Similarmente, la hipótesis implica que ninguna variable ocurre en t_1 ni en t_2. Por hipótesis de inducción, ambos tienen máximo 0, y el máximo de 0 y 0 es 0.

Lema `max_var_form_eq_zero`: Fórmulas sin variables tienen máximo cero

Extiende el resultado a fórmulas:

$$ \forall \varphi \in \text{ArithForm } s,\ (\forall v \in \mathbb{N},\ \text{NotOccurForm } v\ \varphi) \implies \text{max_var_form}(\varphi) = 0 $$

Estructura de la demostración

Por casos sobre la estructura de φ:

- **`bottom`:** `max_var_form` es 0 por definición.

- **`eq t1 t2`, `list_nth_eq t1 t2 t3`:** Se aplica `max_var_term_eq_zero` a cada subtérmino, usando que `NotOccurForm v` para estos constructores implica `NotOccur v` en cada término.

- **`impl f1 f2`, `neg f`:** Se aplica la hipótesis de inducción a las subfórmulas.

- **`forall_form n f`:** La hipótesis `NotOccurForm n (forall_form n f)` implica $n \neq n$, lo cual es una contradicción. Por tanto, este caso es imposible: una fórmula con un cuantificador no puede tener `NotOccurForm` para la variable ligada.

- **`provable_op h t`, `true_op h t`:** Se aplica `max_var_term_eq_zero` al término.

Significado

Estos lemas formalizan la intuición de que una fórmula es una sentencia (no contiene variables en absoluto) exactamente cuando su máximo índice de variable es 0. La dirección demostrada aquí es la implicación "sin variables $\implies$ máximo 0". La dirección recíproca (si el máximo es 0, entonces no hay variables) también es cierta pero no se demuestra explícitamente aquí, ya que la dirección establecida es la que se necesita para las aplicaciones.

Estos resultados son utilizados en la construcción de Henkin-Lindenbaum para verificar que el conjunto inicial Γ satisface la condición de que todas sus fórmulas tienen `max_var_form = 0` (o están acotadas por alguna cota), lo cual es necesario para aplicar `exists_fresh_var_of_deriv` y generar variables frescas. También son relevantes para el lema `substAll_sentencia`, ya que la hipótesis `∀ v, NotOccurForm v φ` implica `max_var_form φ = 0`, y por tanto la condición de `NotFreeInSet_of_max_var_form_zero` se satisface para cualquier $w > 0$.

La imposibilidad del caso `forall_form` en `max_var_form_eq_zero` refleja un hecho fundamental: una fórmula que contiene un cuantificador no puede ser una sentencia en el sentido de no contener ninguna variable en absoluto, ya que la variable ligada es una variable que ocurre (aunque ligada) en la fórmula. La noción relevante para el Teorema de Verdad es la de fórmula sin variables libres, no sin variables en absoluto, pero la condición más fuerte es útil para las construcciones sintácticas.

--
==========

-- SOPORTE INDUCTIVO: LAS SENTENCIAS TIENEN MAX_VAR_FORM = 0

```
--
============================================================================


lemma max_var_term_eq_zero {s : Nat} (t : ArithTerm s) (h : ∀ v, NotOccur v t) :
max_var_term t = 0 :=
  match t with
  | ArithTerm.var n => by
    have h_contra := h n
    cases h_contra with | var _ h_ne => contradiction
  | ArithTerm.const _ => rfl
  | ArithTerm.henkin _ => rfl
  | ArithTerm.zero => rfl
  | ArithTerm.succ t' => by
    have h' : ∀ v, NotOccur v t' := by
      intro v
      have h_all := h v
      cases h_all with | succ _ h_sub => exact h_sub
    exact max_var_term_eq_zero t' h'
  | ArithTerm.add t1 t2 => by
    have h1 : ∀ v, NotOccur v t1 := by
      intro v
      have h_all := h v
      cases h_all with | add _ _ h1' _ => exact h1'
    have h2 : ∀ v, NotOccur v t2 := by
      intro v
      have h_all := h v
      cases h_all with | add _ _ _ h2' => exact h2'
    dsimp [max_var_term]
    rw [max_var_term_eq_zero t1 h1, max_var_term_eq_zero t2 h2]
```

```
  rfl
| ArithTerm.mul t1 t2 => by
  have h1 : ∀ v, NotOccur v t1 := by
    intro v
    have h_all := h v
    cases h_all with | mul _ _ h1' _ => exact h1'
  have h2 : ∀ v, NotOccur v t2 := by
    intro v
    have h_all := h v
    cases h_all with | mul _ _ _ h2' => exact h2'
  dsimp [max_var_term]
  rw [max_var_term_eq_zero t1 h1, max_var_term_eq_zero t2 h2]
  rfl
| ArithTerm.exp t1 t2 => by
  have h1 : ∀ v, NotOccur v t1 := by
    intro v
    have h_all := h v
    cases h_all with | exp _ _ h1' _ => exact h1'
  have h2 : ∀ v, NotOccur v t2 := by
    intro v
    have h_all := h v
    cases h_all with | exp _ _ _ h2' => exact h2'
  dsimp [max_var_term]
  rw [max_var_term_eq_zero t1 h1, max_var_term_eq_zero t2 h2]
  rfl
| ArithTerm.pred t' => by
  have h' : ∀ v, NotOccur v t' := by
    intro v
```

```
    have h_all := h v
    cases h_all with | pred _ h_sub => exact h_sub
  exact max_var_term_eq_zero t' h'

lemma max_var_form_eq_zero {s : Nat} (φ : ArithForm s) (h : ∀ v, NotOccurForm v φ) :
max_var_form φ = 0 :=
 match φ with
 | ArithForm.bottom => rfl
 | ArithForm.eq t1 t2 => by
  have h1 : ∀ v, NotOccur v t1 := by
   intro v
   have h_all := h v
   exact h_all.left
  have h2 : ∀ v, NotOccur v t2 := by
   intro v
   have h_all := h v
   exact h_all.right
  dsimp [max_var_form]
  rw [max_var_term_eq_zero t1 h1, max_var_term_eq_zero t2 h2]
  rfl
 | ArithForm.impl f1 f2 => by
  have h1 : ∀ v, NotOccurForm v f1 := by
   intro v
   have h_all := h v
   exact h_all.left
  have h2 : ∀ v, NotOccurForm v f2 := by
   intro v
   have h_all := h v
```

```
    exact h_all.right
  dsimp [max_var_form]
  rw [max_var_form_eq_zero f1 h1, max_var_form_eq_zero f2 h2]
  rfl
| ArithForm.neg f => by
  have h1 : ∀ v, NotOccurForm v f := by
    intro v
    have h_all := h v
    exact h_all
  dsimp [max_var_form]
  rw [max_var_form_eq_zero f h1]
| ArithForm.forall_form n f => by
  have h_contra := (h n).left
  contradiction
| ArithForm.provable_op heq t => by
  have h1 : ∀ v, NotOccur v t := by
    intro v
    have h_all := h v
    exact h_all
  dsimp [max_var_form]
  rw [max_var_term_eq_zero t h1]
| ArithForm.true_op heq t => by
  have h1 : ∀ v, NotOccur v t := by
    intro v
    have h_all := h v
    exact h_all
  dsimp [max_var_form]
  rw [max_var_term_eq_zero t h1]
```

```
  | ArithForm.list_nth_eq t1 t2 t3 => by
    have h1 : ∀ v, NotOccur v t1 := by
      intro v
      have h_all := h v
      exact h_all.left
    have h2 : ∀ v, NotOccur v t2 := by
      intro v
      have h_all := h v
      exact h_all.right.left
    have h3 : ∀ v, NotOccur v t3 := by
      intro v
      have h_all := h v
      exact h_all.right.right
    dsimp [max_var_form]
    rw [max_var_term_eq_zero t1 h1, max_var_term_eq_zero t2 h2,
  max_var_term_eq_zero t3 h3]
    rfl
```

Lemas de frescura y pertenencia para la secuencia de Henkin

Se demuestran tres lemas que establecen propiedades fundamentales sobre las variables en la lista de testigos y la estructura de la secuencia de Henkin.

Lema `max_var_form_lt_fresh_var_of_GammaWitnessesList`: Cota estricta para variables en la lista

Establece que cualquier fórmula χ en la lista de testigos acumulada hasta el paso m tiene su máximo de variables estrictamente acotado por la variable fresca que se generaría en el paso m:

$$ \forall m \in \mathbb{N},\ \forall \chi \in \text{GammaWitnessesList } m, $$

$$ \text{max_var_form}(\chi) < \max(\text{max_var_list}(\text{GammaWitnessesList } m),\ \text{max_var_form}(\text{decodeExists } m)) + 1 $$

Estructura de la demostración

1. Por `max_var_list_ge`, se tiene $\text{max_var_form}(\chi) \le \text{max_var_list}(\text{GammaWitnessesList } m)$.

2. La cota $\text{max_var_list}(\text{GammaWitnessesList } m)$ está acotada a su vez por el máximo con $\text{max_var_form}(\text{decodeExists } m)$.

3. Sumando 1 se obtiene la desigualdad estricta.

Significado

Este lema garantiza que la variable fresca w generada en el paso m de la construcción de Henkin (definida como ese máximo más uno) es estrictamente mayor

que cualquier variable que aparezca en las fórmulas ya acumuladas. Esto es crucial para asegurar que w es efectivamente fresca respecto a todas las fórmulas procesadas hasta el momento.

Lema `NotFreeInSet_freshVar_of_GammaWitnessesList`: La variable fresca no es libre en la lista

Establece que la variable fresca generada en el paso m no es libre en ninguna fórmula de la lista de testigos acumulada:

$$ \text{sea } w := \max(\text{max_var_list}(\text{GammaWitnessesList } m),\ \text{max_var_form}(\text{decodeExists } m)) + 1 $$

$$ \text{NotFreeInSet } w\ \{\varphi \mid \varphi \in \text{GammaWitnessesList } m\} $$

La demostración es directa: para cada φ en la lista, por el lema anterior $\text{max_var_form}(\varphi) < w$, lo que implica `NotOccurForm w φ` y por tanto `NotFreeForm w φ`.

Lema `mem_GammaSeq`: Descomposición de la pertenencia a la secuencia

Establece que si una fórmula χ pertenece a `GammaSeq Γ m`, entonces o bien χ pertenecía a Γ originalmente, o bien χ es uno de los axiomas de testigo añadidos (es decir, pertenece a `GammaWitnessesList m`):

$$ \chi \in \text{GammaSeq } \Gamma\ m \implies \chi \in \Gamma \lor \chi \in \text{GammaWitnessesList } m $$

Estructura de la demostración

Por inducción sobre m:

- **Caso base ($m = 0$):** `GammaSeq Γ 0 = Γ` y `GammaWitnessesList 0 = []`. La pertenencia está en Γ.

- **Paso inductivo ($m \to m+1$):** Se analiza si se añadió un axioma de testigo:

  - Si no se añadió (`getExistentialComponents` devuelve `none`), la secuencia y la lista no cambian, y se aplica la hipótesis de inducción.

  - Si se añadió un axioma (`some (v, ψ)`), χ puede ser el nuevo axioma (pertenece a la lista) o estar en el conjunto anterior, en cuyo caso se aplica la hipótesis de inducción y se extiende la pertenencia a la lista (si estaba en la lista anterior, sigue estándolo en la extendida).

Significado

Este lema proporciona una caracterización estructural de la secuencia de Henkin: el conjunto `GammaSeq Γ m` está compuesto exactamente por las fórmulas originales de Γ más los axiomas de testigo añadidos hasta el paso m. Es útil para razonar por inducción sobre la construcción: cualquier fórmula en la secuencia o bien es "antigua" (de Γ) o bien es "nueva" (un axioma de testigo). Esta descomposición permite aplicar hipótesis separadas para cada tipo de fórmula en las demostraciones de preservación de consistencia y otras propiedades.

Juntos, estos tres lemas proporcionan el control necesario sobre las variables y la estructura de la secuencia de Henkin para demostrar que la construcción iterativa preserva la consistencia. La variable fresca w está garantizada como no libre en todas las fórmulas acumuladas, lo cual es una condición necesaria para aplicar la regla de generalización universal y los lemas de cambio de variable ligada en los pasos posteriores de la construcción.

```
lemma max_var_form_lt_fresh_var_of_GammaWitnessesList (m : Nat) (χ : ArithForm 0)
(h : χ ∈ GammaWitnessesList m) :

  max_var_form χ < Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form
(decodeExists (s := 0) m)) + 1 := by

 have h_le : max_var_form χ ≤ max_var_list (GammaWitnessesList m) :=
max_var_list_ge (GammaWitnessesList m) χ h

 have h_max : max_var_list (GammaWitnessesList m) ≤ Nat.max (max_var_list
(GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) :=
Nat.le_max_left _ _

 omega


lemma NotFreeInSet_freshVar_of_GammaWitnessesList (m : Nat) :

  let w := Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form
(decodeExists (s := 0) m)) + 1

  NotFreeInSet w {φ | φ ∈ GammaWitnessesList m} := by

 intro w φ hφ

 have h_mem : φ ∈ GammaWitnessesList m := hφ

 have h_lt : max_var_form φ < w :=
max_var_form_lt_fresh_var_of_GammaWitnessesList m φ h_mem

 have h_not_occ : NotOccurForm w φ := NotOccurForm_of_gt_max_var φ w (by omega)

 exact NotFreeForm_of_NotOccurForm w φ h_not_occ
```

```
lemma mem_GammaSeq {Γ : Set (ArithForm 0)} {m : Nat} {χ : ArithForm 0} :
  χ ∈ GammaSeq Γ m → χ ∈ Γ ∨ χ ∈ GammaWitnessesList m := by
 induction m with
 | zero =>
  unfold GammaSeq GammaWitnessesList
  intro h
  left; exact h
 | succ m ih =>
  dsimp [GammaSeq, GammaWitnessesList]
  generalize h_dec : getExistentialComponents (decodeExists (s := 0) m) = opt
  cases opt with
  | none =>
   intro h
   exact ih h
  | some pair =>
   rcases pair with ⟨v, ψ⟩
   intro h
   simp only [Set.mem_insert_iff] at h
   rcases h with rfl | h_old
   · right
    simp only [List.mem_cons, true_or]
   · rcases ih h_old with h_Γ | h_list
    · left; exact h_Γ
    · right
     simp only [List.mem_cons, h_list, or_true]
```

Lema `NotFreeInSet_freshVar_of_GammaSeq`: Frescura de la variable fresca en toda la secuencia

Se demuestra un resultado que unifica y extiende los lemas anteriores: la variable fresca generada en el paso m de la construcción de Henkin no es libre en ninguna fórmula del conjunto `GammaSeq Γ m` (que incluye tanto las fórmulas originales de Γ como los axiomas de testigo añadidos).

Enunciado formal

$$ \forall \Gamma \subseteq \text{ArithForm } 0,\ (\forall \psi \in \Gamma,\ \text{max_var_form}(\psi) = 0) \implies \forall m \in \mathbb{N}, $$

$$ \text{sea } w := \max(\text{max_var_list}(\text{GammaWitnessesList } m),\ \text{max_var_form}(\text{decodeExists } m)) + 1, $$

$$ \text{NotFreeInSet } w\ (\text{GammaSeq } \Gamma\ m) $$

Estructura de la demostración

Para una fórmula $\chi \in \text{GammaSeq } \Gamma\ m$, se aplica el lema `mem_GammaSeq` para descomponer su pertenencia en dos casos:

Caso 1: $\chi \in \Gamma$ (fórmula original). Por la hipótesis sobre Γ, $\text{max_var_form}(\chi) = 0$. Como $w \ge 1$ (es un máximo más uno), se tiene

$w > \text{max_var_form}(\chi)$. Por `NotOccurForm_of_gt_max_var`, w no ocurre en χ, y por `NotFreeForm_of_NotOccurForm`, w no es libre en χ.

Caso 2: $\chi \in \text{GammaWitnessesList } m$ (axioma de testigo). Se aplica directamente el lema `NotFreeInSet_freshVar_of_GammaWitnessesList`, que establece precisamente que w no es libre en ninguna fórmula de la lista de testigos.

Significado

Este lema es un resultado de uniformidad fundamental para la construcción de Henkin: garantiza que la variable fresca w elegida en el paso m es simultáneamente no libre en todas las fórmulas acumuladas hasta ese momento, tanto las originales como las añadidas. Esta propiedad de frescura global es exactamente lo que se necesita para aplicar la regla de generalización universal (`forall_intro`) en los pasos posteriores de la construcción, así como para los lemas de cambio de variable ligada y alpha-conversión.

La hipótesis de que todas las fórmulas originales de Γ tienen `max_var_form = 0` (es decir, son sentencias sin variables) es una condición que se satisface típicamente en las teorías iniciales consideradas (por ejemplo, conjuntos de axiomas cerrados). Esta condición asegura que el caso base de la construcción ya tiene la propiedad de frescura deseada, y el lema muestra que esta propiedad se mantiene inductivamente a lo largo de toda la construcción de Henkin.

La capacidad de generar variables frescas que no son libres en todo el conjunto acumulado es una de las piedras angulares del método de Henkin. Sin esta garantía, las extensiones sucesivas de la teoría podrían introducir interferencias entre las variables ligadas de los axiomas de testigo y las variables libres de otras fórmulas, comprometiendo la corrección de la construcción. Este lema, junto con su contraparte para constantes de Henkin (`ConstantNotOccur_above_max_Henkin` y lemas relacionados), proporciona la base sintáctica sólida sobre la que se asienta toda la maquinaria de completitud.

```
lemma NotFreeInSet_freshVar_of_GammaSeq (Γ : Set (ArithForm 0)) (hΓ : ∀ ψ ∈ Γ, max_var_form ψ = 0) (m : Nat) :
  let w := Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) + 1
  NotFreeInSet w (GammaSeq Γ m) := by
 intro w χ hχ
 rcases mem_GammaSeq hχ with h_Γ | h_list
 · have h_zero := hΓ χ h_Γ
  have h_not_occ : NotOccurForm w χ := by
   apply NotOccurForm_of_gt_max_var
   rw [h_zero]
   omega
  exact NotFreeForm_of_NotOccurForm w χ h_not_occ
 · have h_list' : χ ∈ {φ | φ ∈ GammaWitnessesList m} := h_list
  have h_fresh := NotFreeInSet_freshVar_of_GammaWitnessesList m
  exact h_fresh χ h_list'
```

Lema `NotFreeInSet_GammaSeq_of_ge`: Frescura generalizada para cotas superiores

Se demuestra una generalización del lema anterior que establece que cualquier variable z mayor o igual que la variable fresca generada en el paso m tampoco es libre en el conjunto `GammaSeq Γ m`. Este resultado proporciona flexibilidad en la elección de variables frescas, permitiendo tomar cualquier variable suficientemente grande.

Enunciado formal

$$ \forall \Gamma \subseteq \text{ArithForm } 0,\ (\forall \psi \in \Gamma,\ \text{max_var_form}(\psi) = 0) \implies \forall m, z \in \mathbb{N}, $$

$$ z \ge \max(\text{max_var_list}(\text{GammaWitnessesList } m),\ \text{max_var_form}(\text{decodeExists } m)) + 1 \implies \text{NotFreeInSet } z\ (\text{GammaSeq } \Gamma\ m) $$

Estructura de la demostración

Para una fórmula $\chi \in \text{GammaSeq } \Gamma\ m$, se descompone su pertenencia mediante `mem_GammaSeq`:

Caso 1: $\chi \in \Gamma$. Por hipótesis, $\text{max_var_form}(\chi) = 0$. Como $z \ge 1$ (es mayor o igual que un máximo más uno, y el máximo es al menos 0), se tiene $z > 0 = \text{max_var_form}(\chi)$. Por `NotOccurForm_of_gt_max_var`, z no ocurre en χ, y por `NotFreeForm_of_NotOccurForm`, z no es libre en χ.

Caso 2: $\chi \in \text{GammaWitnessesList } m$. Por `max_var_list_ge`, se tiene $\text{max_var_form}(\chi) \le \text{max_var_list}(\text{GammaWitnessesList } m)$. Este último está acotado por el máximo con `max_var_form(decodeExists m)`. Sumando 1 y usando la hipótesis $z \ge \text{ese máximo} + 1$, se deduce $z >$

\text{max_var_form}(\chi)$. Nuevamente, por `NotOccurForm_of_gt_max_var` y `NotFreeForm_of_NotOccurForm`, z no es libre en χ.

Significado

Este lema establece una propiedad de monotonía para la frescura: si una variable es suficientemente grande (mayor o igual que la cota calculada), entonces es fresca respecto al conjunto. Esto es análogo a la propiedad de que si $w > \text{max_var_form}(\varphi)$, entonces w no ocurre en φ; aquí se extiende a conjuntos completos y a la noción de "no ser libre".

La importancia de esta generalización radica en que, en la construcción iterativa de Henkin-Lindenbaum, a menudo se necesita elegir variables frescas en pasos posteriores que no solo sean mayores que las variables del paso actual, sino también mayores que variables introducidas en pasos intermedios. La monotonía garantiza que cualquier variable mayor que la cota del paso m también es fresca para ese paso. Esto permite "acumular" cotas a lo largo de la construcción: si se toma el máximo de las cotas de varios pasos, cualquier variable mayor que ese máximo es fresca para todos ellos simultáneamente.

Esta propiedad es utilizada en la demostración de `exists_fresh_var_of_deriv` y en la construcción de Lindenbaum, donde se necesita una variable que sea fresca respecto a un conjunto finito de fórmulas (extraído de algún paso de la secuencia). La capacidad de elegir cualquier variable suficientemente grande, en lugar de una cota exacta, proporciona la flexibilidad necesaria para coordinar múltiples condiciones de frescura simultáneamente.

--
==========

```
-- NUEVO LEMA DE FRESCURA GENERALIZADA PARA CO-DOMINIOS DE COTAS
--
================================================================


lemma NotFreeInSet_GammaSeq_of_ge (Γ : Set (ArithForm 0)) (hΓ : ∀ ψ ∈ Γ, max_var_form ψ = 0) (m : Nat) (z : Nat)
  (hz : z ≥ Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) + 1) :
  NotFreeInSet z (GammaSeq Γ m) := by
 intro χ hχ
 rcases mem_GammaSeq hχ with h_Γ | h_list
 · have h_zero := hΓ χ h_Γ
   have h_not_occ : NotOccurForm z χ := by
    apply NotOccurForm_of_gt_max_var
    rw [h_zero]
    omega
   exact NotFreeForm_of_NotOccurForm z χ h_not_occ
 · have h_le := max_var_list_ge (GammaWitnessesList m) χ h_list
   have h_max_le : max_var_list (GammaWitnessesList m) ≤ Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) := Nat.le_max_left _ _
   have h_gt : z > max_var_form χ := by omega
   have h_not_occ := NotOccurForm_of_gt_max_var χ z h_gt
   exact NotFreeForm_of_NotOccurForm z χ h_not_occ
```

Teorema `GammaSeq_consistent`: Consistencia de la secuencia de Henkin

Se demuestra el teorema fundamental de la Fase 16a: la construcción iterativa de la secuencia de Henkin preserva la consistencia en cada paso. Este es el resultado central que garantiza que la extensión por testigos de Henkin no introduce contradicciones.

Enunciado formal

$$ \forall \Gamma \subseteq \text{ArithForm } 0, $$

$$ (\forall \varphi \in \Gamma,\ \text{maxHenkinForm } \varphi = 0) \land (\forall \varphi \in \Gamma,\ \text{max_var_form } \varphi = 0) \land \text{Consistent } \Gamma \implies $$

$$ \forall n \in \mathbb{N},\ \text{Consistent}(\text{GammaSeq } \Gamma\ n) $$

junto con la hipótesis adicional de que en cada paso m donde se procesa una fórmula existencial, la variable fresca generada no es libre en el conjunto acumulado.

Estructura de la demostración

Por inducción sobre n:

Caso base ($n = 0$): `GammaSeq Γ 0 = Γ`, que es consistente por hipótesis.

Paso inductivo ($m \to m+1$): Se analiza la definición de `GammaSeq` en el paso $m+1$:

- **Caso `none`:** No se añade ninguna fórmula, la consistencia se hereda de la hipótesis de inducción.

- **Caso `some (v, ψ)`:** Se procesa una fórmula existencial. Se supone, por contradicción, que el conjunto extendido con el axioma de testigo es inconsistente, es decir, existe una derivación de $\bot$ a partir de $\Gamma' \cup \{A \to B\}$, donde $\Gamma' = \text{GammaSeq } \Gamma\ m$, $A = \text{freshEx}$ y $B = \psi[v := \text{henkin } c]$.

La demostración procede en varias etapas:

1. **Extracción de información:** De la derivación de $\bot$ en el conjunto extendido, por `neg_intro` y los lemas `imp_neg_elim_left/right`, se obtienen derivaciones de A y $\neg B$ en Γ'.

2. **Preparación para eliminación de constantes:** Se toma una derivación de $\bot$ en $\Gamma' \cup \{B\}$ (que existe porque $\Gamma' \vdash \neg B$). Por `SPA_Deduction_Internal_finite_support`, existe un subconjunto finito $\Gamma_0 \subseteq \Gamma' \cup \{B\}$ que prueba $\bot$.

3. **Elección de variable fresca z:** Se calcula una cota M que supera tanto las variables en Γ_0 como la variable fresca w del paso actual. Por `exists_fresh_var_of_deriv`, existe $z > M$ que es fresca para la derivación y no es libre en Γ_0.

4. **Eliminación de la constante de Henkin:** Se aplica `substHenkin_preservation_of_fresh` a la derivación, reemplazando la constante c por la variable fresca z. Se verifica que el conjunto transformado está contenido en $\Gamma' \cup \{\psi[v := \text{var } z]\}$, utilizando que las fórmulas de Γ' no contienen a c (por la cota `GammaSeq_maxHenkin_bound`) y que B se transforma adecuadamente (por los lemas de conmutación `substHenkinForm_substForm_henkin_comm` y `substHenkinForm_fresh`).

5. **Reescritura de la fórmula:** Se utiliza el lema `substForm_substForm_var_comm` para reescribir $\psi[v := \text{var } z]$ como $\psi_w[w := \text{var } z]$, donde $\psi_w = \psi[v := \text{var } w]$.

6. **Aplicación de `alpha_generalize`:** Se verifican las hipótesis del teorema `alpha_generalize`:

- z no es libre en Γ' (por `NotFreeInSet_GammaSeq_of_ge`).

- z no ocurre en ψ_w (por `NotOccurForm_of_gt_max_var`, ya que $z > w \ge \text{max_var_form}(\psi_w)$).

- w no es libre en Γ' (por la hipótesis `h_fresh_vars`).

- w no está ligada en ψ_w (por `NoBoundVar_substForm_var`, ya que w es fresca respecto a ψ).

Se obtiene $\Gamma' \vdash \forall w\ \neg \psi_w$, es decir, $\Gamma' \vdash \neg A$.

7. **Contradicción:** Pero en el paso 1 se obtuvo $\Gamma' \vdash A$. La presencia simultánea de A y $\neg A$ en Γ' contradice la consistencia de Γ' (hipótesis de inducción).

Significado

`GammaSeq_consistent` es el teorema que cierra la construcción de Henkin, estableciendo que el proceso iterativo de añadir axiomas de testigo preserva la consistencia. Es la culminación de una larga cadena de lemas que incluyen:

- Control de cotas para constantes de Henkin (`maxHenkinTerm_substTerm`, `GammaSeq_maxHenkin_bound`)

- Control de cotas para variables (`max_var_term_substTerm`, `max_var_form_lt_fresh_var_of_GammaWitnessesList`)

- Generación de variables frescas (`exists_fresh_var_of_deriv`)

- Preservación de la demostrabilidad bajo eliminación de constantes (`substHenkin_preservation_of_fresh`)

- Alpha-generalización (`alpha_generalize`)

La demostración utiliza de manera esencial la maquinaria de frescura y eliminación de constantes de Henkin para mostrar que cualquier inconsistencia en la teoría extendida implicaría una inconsistencia en la teoría original, lo cual es imposible por hipótesis. Este es exactamente el argumento estándar de Henkin: los axiomas de testigo no introducen contradicciones porque las nuevas constantes pueden ser eliminadas reemplazándolas por variables frescas, y los axiomas resultantes se vuelven demostrables a partir de la teoría original.

Con este teorema, junto con `GammaLimit_consistent` (que extiende la consistencia al límite), se completa la demostración de que el límite de Henkin es una extensión consistente de la teoría original, preparando el terreno para la construcción de Lindenbaum y, finalmente, para la prueba de completitud.

```
--
============================================================

-- REESCRITURA CONSTRUCTIVA DE GammaSeq_consistent (SIN AXIOMAS)

--
============================================================


theorem GammaSeq_consistent (Γ : Set (ArithForm 0))

  (h_initial_c : ∀ φ ∈ Γ, maxHenkinForm φ = 0)

  (h_initial_var : ∀ φ ∈ Γ, max_var_form φ = 0)

  (h_fresh_vars : ∀ m v ψ, getExistentialComponents (decodeExists (s := 0) m) = some (v, ψ) →
```

```
    let w := Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) + 1;
    NotFreeInSet w (GammaSeq Γ m))
  (h_con : Consistent Γ) :
  ∀ n, Consistent (GammaSeq Γ n) := by
 intro n
 induction n with
 | zero => exact h_con
 | succ m ih =>
  unfold GammaSeq
  split
  · rename_i v ψ h_dec
    let w := Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) + 1
    let freshEx := ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)))
    let c : HenkinConst := ⟨Nat.max (maxHenkinList (GammaWitnessesList m)) (maxHenkinForm freshEx) + 1⟩
    intro h_bot
    let Γ' := GammaSeq Γ m
    let A := freshEx
    let B := substForm v (ArithTerm.henkin c) ψ
    have h_neg_impl : SPA_Deduction_Internal 0 Γ' (ArithForm.neg (ArithForm.impl A B)) :=
      SPA_Deduction_Internal.neg_intro Γ' (ArithForm.impl A B) h_bot
    have h_A : SPA_Deduction_Internal 0 Γ' A :=
      imp_neg_elim_left Γ' A B h_neg_impl
    have h_neg_B : SPA_Deduction_Internal 0 Γ' (ArithForm.neg B) :=
      imp_neg_elim_right Γ' A B h_neg_impl
    have h_bot_B : SPA_Deduction_Internal 0 (insert B Γ') ArithForm.bottom := by
```

```
      have h_B : SPA_Deduction_Internal 0 (insert B Γ') B := SPA_Deduction_Internal.hyp
_ B (Or.inl rfl)

      have h_neg_B' : SPA_Deduction_Internal 0 (insert B Γ') (ArithForm.neg B) :=

       SPA_Deduction_Internal.weaken Γ' (ArithForm.neg B) (insert B Γ') h_neg_B
(Set.subset_insert _ _)

      exact SPA_Deduction_Internal.neg_elim _ B h_B h_neg_B'

    obtain ⟨Γ0, h_sub, h_ded⟩ := SPA_Deduction_Internal_finite_support h_bot_B

    let M := Nat.max (max_var_set Γ0) w

    have h_Γ0 : ∀ ψ' ∈ (Γ0 : Set (ArithForm 0)), max_var_form ψ' ≤ M := by

      intro ψ' hψ'

      have h_le := max_var_form_le_max_var_set Γ0 ψ' hψ'

      exact Nat.le_trans h_le (Nat.le_max_left _ _)

    have h_bottom : max_var_form (ArithForm.bottom (s := 0)) ≤ M := Nat.zero_le _

    obtain ⟨z, hz_gt, h_fresh_z, h_not_free_z⟩ := exists_fresh_var_of_deriv h_ded M h_Γ0
h_bottom

    have h_subst_henkin := substHenkin_preservation_of_fresh (Γ0 : Set (ArithForm 0))
ArithForm.bottom h_ded z h_fresh_z c

    have h_sub_set : {x | ∃ ψ' ∈ (Γ0 : Set (ArithForm 0)), substHenkinForm c
(ArithTerm.var z) ψ' = x} ⊆

                insert (substForm v (ArithTerm.var z) ψ) Γ' := by

      rintro x ⟨ψ', hψ', rfl⟩

      have h_in_or := h_sub hψ'

      cases h_in_or with

      | inl h_eq_B =>

       subst h_eq_B

       left

       rw [substHenkinForm_substForm_henkin_comm]

       have h_max_freshEx : maxHenkinForm freshEx = maxHenkinForm ψ := by

         dsimp [freshEx, maxHenkinForm]

         rw [maxHenkinForm_substForm_var ψ v w]
```

```
      have h_co : ConstantNotOccurForm c ψ := by
        apply ConstantNotOccurForm_of_gt_maxHenkinForm
        dsimp [c]
        have h_le : maxHenkinForm freshEx ≤ Nat.max (maxHenkinList (GammaWitnessesList m)) (maxHenkinForm freshEx) := Nat.le_max_right _ _
        rw [h_max_freshEx] at h_le ⊢
        omega
      rw [substHenkinForm_fresh ψ c (ArithTerm.var z) h_co]
    | inr h_in_Gamma =>
      right
      have h_co : ConstantNotOccurForm c ψ' := by
        apply ConstantNotOccurForm_of_gt_maxHenkinForm
        dsimp [c]
        have h_le1 := GammaSeq_maxHenkin_bound Γ h_initial_c m ψ' h_in_Gamma
        have h_le2 : maxHenkinList (GammaWitnessesList m) ≤ Nat.max (maxHenkinList (GammaWitnessesList m)) (maxHenkinForm freshEx) := Nat.le_max_left _ _
        have h_trans := Nat.le_trans h_le1 h_le2
        omega
      rw [substHenkinForm_fresh ψ' c (ArithTerm.var z) h_co]
      exact h_in_Gamma
  have h_bot_z : SPA_Deduction_Internal 0 (insert (substForm v (ArithTerm.var z) ψ) Γ') ArithForm.bottom :=
    SPA_Deduction_Internal.weaken _ _ _ h_subst_henkin h_sub_set
  have h_subst_self : substForm v (ArithTerm.var z) ψ = substForm w (ArithTerm.var z) (substForm v (ArithTerm.var w) ψ) := by
    have h_max_le : max_var_form ψ ≤ max_var_form (decodeExists (s := 0) m) := by
      rw [decodeExists_eq_of_getExistentialComponents_some m v ψ h_dec]
      dsimp [max_var_form]
      exact Nat.le_max_right v (max_var_form ψ)
```

```
      have hw_gt : w > max_var_form ψ := by
       have h_max_right : max_var_form (decodeExists (s := 0) m) ≤
                    Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form
(decodeExists (s := 0) m)) :=
        Nat.le_max_right _ _
       have hw_ge : w > max_var_form (decodeExists (s := 0) m) := by
        dsimp [w]
        omega
       exact Nat.lt_of_le_of_lt h_max_le hw_ge
      have hw_not_occ := NotOccurForm_of_gt_max_var ψ w hw_gt
      rw [← substForm_substForm_var_comm ψ v w z hw_not_occ]
    rw [h_subst_self] at h_bot_z
    let ψ_w := substForm v (ArithTerm.var w) ψ
    have h_neg_z : SPA_Deduction_Internal 0 Γ' (ArithForm.neg (substForm w
(ArithTerm.var z) ψ_w)) :=
     SPA_Deduction_Internal.neg_intro Γ' (substForm w (ArithTerm.var z) ψ_w)
h_bot_z
    have h_fresh_Γ' : NotFreeInSet z Γ' := by
     apply NotFreeInSet_GammaSeq_of_ge Γ h_initial_var m z
     have hz_ge_w : z ≥ w := by
      clear h_sub h_ded h_Γ0 h_bottom h_fresh_z h_not_free_z h_subst_henkin
h_sub_set h_bot_z h_subst_self h_neg_z
       have h_max_right_M : w ≤ Nat.max (max_var_set Γ0) w := Nat.le_max_right _ _
       dsimp [M] at hz_gt ⊢
       omega
     exact hz_ge_w
    have h_fresh_ψ_w : NotOccurForm z ψ_w := by
     apply NotOccurForm_of_gt_max_var
     have h_max_le : max_var_form ψ ≤ max_var_form (decodeExists (s := 0) m) := by
      rw [decodeExists_eq_of_getExistentialComponents_some m v ψ h_dec]
```

```
      dsimp [max_var_form]
      exact Nat.le_max_right v (max_var_form ψ)
    have hw_ge : w > max_var_form (decodeExists (s := 0) m) := by
      have h_max_right : max_var_form (decodeExists (s := 0) m) ≤
            Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) :=
       Nat.le_max_right _ _
      dsimp [w]
      omega
    have hw_gt : w > max_var_form ψ := Nat.lt_of_le_of_lt h_max_le hw_ge
    have h_le : max_var_form (substForm v (ArithTerm.var (s := 0) w) ψ) ≤
          Nat.max (max_var_form ψ) (max_var_term (ArithTerm.var (s := 0) w)) := by
      have h_raw := max_var_form_substForm ψ v (ArithTerm.var (s := 0) w)
      exact h_raw
    have h_var_term : max_var_term (ArithTerm.var (s := 0) w) = w := rfl
    have h_max_ψ_w : max_var_form ψ_w ≤ w := by
      apply Nat.le_trans h_le
      rw [h_var_term]
      exact Nat.max_le.mpr ⟨Nat.le_of_lt hw_gt, Nat.le_refl w⟩
    have : z > w := by
      clear h_sub h_ded h_Γ0 h_bottom h_fresh_z h_not_free_z h_subst_henkin h_sub_set h_bot_z h_subst_self h_neg_z h_fresh_Γ'
      have h_max_right_M : w ≤ Nat.max (max_var_set Γ0) w := Nat.le_max_right _ _
      dsimp [M] at hz_gt ⊢
      omega
    exact Nat.lt_of_le_of_lt h_max_ψ_w this
  have h_fresh_v : NotFreeInSet w Γ' := h_fresh_vars m v ψ h_dec
  have h_v_bound : NoBoundVar w ψ_w := by
    have h_max_le : max_var_form ψ ≤ max_var_form (decodeExists (s := 0) m) := by
```

```
      rw [decodeExists_eq_of_getExistentialComponents_some m v ψ h_dec]
      dsimp [max_var_form]
      exact Nat.le_max_right v (max_var_form ψ)
    have hw_ge : w > max_var_form (decodeExists (s := 0) m) := by
      have h_max_right : max_var_form (decodeExists (s := 0) m) ≤
                  Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form
(decodeExists (s := 0) m)) :=
        Nat.le_max_right _ _
      dsimp [w]
      omega
    have hw_gt : w > max_var_form ψ := Nat.lt_of_le_of_lt h_max_le hw_ge
    have hw_not_occ := NotOccurForm_of_gt_max_var ψ w hw_gt
    have h_nobound := NoBoundVar_of_NotOccurForm w ψ hw_not_occ
    exact NoBoundVar_substForm_var ψ v w h_nobound
  have h_forall := alpha_generalize Γ' w z ψ_w h_fresh_Γ' h_fresh_ψ_w h_fresh_v
h_v_bound h_neg_z
  have h_bot_final := SPA_Deduction_Internal.neg_elim Γ' (ArithForm.forall_form w
(ArithForm.neg ψ_w)) h_forall h_A
  exact ih h_bot_final
· rename_i _h_none
  exact ih
```

Teorema `GammaSeq_fresh_vars_singleton`: Frescura para el conjunto inicial unitario

Se demuestra que la hipótesis sobre frescura de variables requerida por `GammaSeq_consistent` se satisface automáticamente cuando el conjunto inicial Γ consiste en una única fórmula que es una sentencia (sin variables). Este resultado elimina la necesidad de postular la hipótesis `h_fresh_vars` como un axioma adicional.

Enunciado formal

$$ \forall \varphi \in \text{ArithForm } 0,\ (\forall v \in \mathbb{N},\ \text{NotOccurForm } v\ \varphi) \implies \forall m, v \in \mathbb{N},\ \forall \psi \in \text{ArithForm } 0, $$

$$ \text{getExistentialComponents}(\text{decodeExists } m) = \text{some}(v, \psi) \implies $$

$$ \text{sea } w := \max(\text{max_var_list}(\text{GammaWitnessesList } m),\ \text{max_var_form}(\text{decodeExists } m)) + 1, $$

$$ \text{NotFreeInSet } w\ (\text{GammaSeq } \{\neg\varphi\}\ m) $$

Estructura de la demostración

La prueba es notablemente concisa gracias al lema `NotFreeInSet_freshVar_of_GammaSeq` demostrado anteriormente. Se reduce a verificar la hipótesis de dicho lema:

1. **El conjunto inicial es $\{\neg\varphi\}$:** Se debe verificar que toda fórmula en este conjunto tiene `max_var_form = 0`.

2. **φ es una sentencia:** La hipótesis `∀ v, NotOccurForm v φ` implica, por el lema `max_var_form_eq_zero`, que $\text{max_var_form}(\varphi) = 0$.

3. **La negación preserva el máximo:** $\text{max_var_form}(\neg\varphi) = \text{max_var_form}(\varphi) = 0$.

4. **Aplicación directa:** Con esta verificación, `NotFreeInSet_freshVar_of_GammaSeq` proporciona exactamente la conclusión buscada.

Significado

Este teorema es el eslabón final que cierra la cadena de la construcción de Henkin. Establece que para una teoría inicial consistente en una sola fórmula sentencia (como sería el caso típico de una fórmula que se quiere demostrar satisfacible), la condición de frescura de variables requerida por `GammaSeq_consistent` se cumple automáticamente. Esto significa que no hay necesidad de asumir hipótesis adicionales sobre la frescura de variables en la construcción: la propiedad se sigue de la estructura misma de la construcción y de la hipótesis de que la fórmula inicial es una sentencia.

Combinando `GammaSeq_consistent` con `GammaSeq_fresh_vars_singleton`, se obtiene que para cualquier sentencia φ tal que $\{\neg\varphi\}$ es consistente, la secuencia de Henkin construida a partir de $\{\neg\varphi\}$ es consistente en todos sus pasos. Por `GammaLimit_consistent`, el límite también es consistente. Luego, por la construcción de Lindenbaum, existe una extensión maximal consistente Δ que contiene $\neg\varphi$ y tiene la propiedad de Henkin. Finalmente, por el Teorema de Verdad, el modelo canónico de Δ satisface $\neg\varphi$, y por tanto no satisface φ.

Este es exactamente el argumento de completitud: si φ no es demostrable (es decir, $\{\neg\varphi\}$ es consistente), entonces φ no es universalmente válida (existe un modelo donde es falsa). Equivalentemente, si φ es universalmente válida, entonces φ es demostrable. La demostración de `GammaSeq_fresh_vars_singleton` es la pieza que permite aplicar toda la maquinaria de Henkin-Lindenbaum al caso particular de una teoría formada por una única sentencia, que es el punto de partida de la prueba de completitud. Su simplicidad (una

vez establecidos los lemas previos) refleja la madurez de la infraestructura desarrollada.

```
--
============================================================

-- ELIMINACIÓN DE AXIOMA: `GammaSeq_fresh_vars_singleton` AHORA ES UN TEOREMA DEMOSTRADO

--
============================================================


theorem GammaSeq_fresh_vars_singleton {φ : ArithForm 0} (h_sentencia : ∀ v, NotOccurForm v φ) (m : Nat) (v : Nat) (ψ : ArithForm 0)

  (_h_dec : getExistentialComponents (decodeExists (s := 0) m) = some (v, ψ)) :

  let w := Nat.max (max_var_list (GammaWitnessesList m)) (max_var_form (decodeExists (s := 0) m)) + 1

  NotFreeInSet w (GammaSeq ({ArithForm.neg φ} : Set (ArithForm 0)) m) := by

 have h_zero : ∀ ψ' ∈ ({ArithForm.neg φ} : Set (ArithForm 0)), max_var_form ψ' = 0 := by

  intro ψ' hψ'

  rcases hψ' with rfl

  exact max_var_form_eq_zero φ h_sentencia

 exact NotFreeInSet_freshVar_of_GammaSeq ({ArithForm.neg φ} : Set (ArithForm 0)) h_zero m
```

Lemas de preservación de no ligadura y no libertad bajo sustituciones específicas

Se demuestran tres lemas que establecen propiedades de preservación de frescura bajo operaciones de sustitución particulares. Estos resultados son variantes y generalizaciones de lemas anteriores, adaptados para situaciones específicas que aparecen en la construcción de Henkin y en el Teorema de Verdad.

Lema `NoBoundVar_substForm_general`: Preservación de no ligadura bajo cualquier sustitución

Establece que la propiedad de que una variable z no está ligada en una fórmula φ se preserva bajo cualquier sustitución de una variable v por un término t:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \forall z \in \mathbb{N}, $$

$$ \text{NoBoundVar } z\ \varphi \implies \text{NoBoundVar } z\ (\text{substForm } v\ t\ \varphi) $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos sin cuantificadores:** `NoBoundVar` es siempre verdadero, por lo que se preserva trivialmente.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Si $v = n$, la sustitución no penetra y la fórmula no cambia, preservando `NoBoundVar`. Si $v \neq n$, la sustitución atraviesa el cuantificador. La hipótesis `NoBoundVar z (forall_form n f)` proporciona $z \neq n$ y `NoBoundVar z f`. La sustitución no modifica la variable ligada n, por lo que $z \neq n$ se mantiene, y la hipótesis de inducción garantiza `NoBoundVar z` en el cuerpo sustituido.

Significado

Este lema generaliza `NoBoundVar_substForm_var` (que era el caso particular $t = \text{var } w$) a cualquier término t. Establece que la sustitución de una variable por un término nunca introduce nuevas ligaduras: si una variable no estaba ligada originalmente, no puede pasar a estar ligada después de la sustitución. Esto es intuitivamente claro porque la sustitución solo afecta a las ocurrencias libres de v, no a los cuantificadores. La propiedad es utilizada en las demostraciones de alpha-conversión y en la verificación de las condiciones de `alpha_generalize`.

Lema `NotOccur_substTerm_self_var`: No ocurrencia de v tras sustituir v por w

Establece que si $v \neq w$, entonces v no ocurre en el resultado de sustituir v por $\text{var } w$ en cualquier término t:

$$ \forall t \in \text{ArithTerm } s,\ \forall v, w \in \mathbb{N},\ v \neq w \implies \text{NotOccur } v\ (\text{substTerm } v\ (\text{var } w)\ t) $$

La demostración es por inducción sobre t. El caso `var n` distingue si $n = v$ (se reemplaza por $\text{var } w$, y $v \neq w$ garantiza que v no ocurre) o $n \neq v$ (se preserva, y $v \neq n$).

Lema `NotFreeForm_subst_var_self`: Variable v no es libre tras sustituir v por w

Extiende el lema anterior a fórmulas: si $v \neq w$, entonces v no es libre en $\psi[v := \text{var } w]$:

$$ \forall \psi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ v \neq w \implies \text{NotFreeForm } v\ (\text{substForm } v\ (\text{var } w)\ \psi) $$

La demostración es por inducción sobre ψ, utilizando `NotOccur_substTerm_self_var` para los casos con términos. El caso `forall_form n f` distingue si $v = n$ (la sustitución no penetra, y v está ligada por el cuantificador, por lo que no es libre) o $v \neq n$ (la sustitución se propaga, y por hipótesis de inducción v no es libre en el cuerpo).

Significado

Estos dos últimos lemas formalizan una propiedad crucial para la alpha-conversión: al renombrar una variable libre v a una variable fresca w, la variable original v desaparece completamente de la fórmula (no queda libre ni ocurre). Esto es lo que permite "descargar" variables en las reglas de generalización: después de sustituir v por w y generalizar sobre w, la variable v ya no aparece, cumpliendo la condición de que la variable generalizada no es libre en las hipótesis. Son utilizados en `alpha_generalize` y en la construcción de testigos de Henkin, donde se necesita garantizar que las variables originales no interfieren después del renombramiento.

```
--
================================================================
-- PROPIEDAD DE PRESERVACIÓN DE COTAS DE LIGADURA BAJO SUSTITUCIÓN
--
================================================================


lemma NoBoundVar_substForm_general {s : Nat} (φ : ArithForm s) (v : Nat) (t :
ArithTerm s) (z : Nat) (h : NoBoundVar z φ) :
   NoBoundVar z (substForm v t φ) := by
 induction φ generalizing v t z with
 | bottom => trivial
 | eq _ _ => trivial
 | impl f1 f2 ih1 ih2 =>
   dsimp [substForm, NoBoundVar] at h ⊢
   exact ⟨ih1 v t z h.1, ih2 v t z h.2⟩
 | neg f ih =>
   dsimp [substForm, NoBoundVar] at h ⊢
   exact ih v t z h
 | forall_form n f ih =>
   dsimp [substForm]
   split_ifs with h_vn
   · dsimp [NoBoundVar] at h ⊢
     exact h
   · dsimp [NoBoundVar] at h ⊢
     exact ⟨h.1, ih v t z h.2⟩
 | provable_op _ _ => trivial
 | true_op _ _ => trivial
 | list_nth_eq _ _ _ => trivial
```

```
--
================================================================

-- PROPIEDADES AUXILIARES DE NO-OCURRENCIA Y FRESCURA DE VARIABLES

--
================================================================


lemma NotOccur_substTerm_self_var {s : Nat} (t : ArithTerm s) (v w : Nat) (h_vw : v ≠
w) :

   NotOccur v (substTerm v (ArithTerm.var w) t) := by

 induction t with

 | var n =>

   unfold substTerm

   split_ifs with h_vn

   · exact NotOccur.var w h_vw

   · exact NotOccur.var n h_vn

 | const c => exact NotOccur.const c

 | henkin c => exact NotOccur.henkin c

 | zero => exact NotOccur.zero

 | succ t' ih => exact NotOccur.succ _ ih

 | add t1 t2 ih1 ih2 => exact NotOccur.add _ _ ih1 ih2

 | mul t1 t2 ih1 ih2 => exact NotOccur.mul _ _ ih1 ih2

 | exp t1 t2 ih1 ih2 => exact NotOccur.exp _ _ ih1 ih2

 | pred t' ih => exact NotOccur.pred _ ih


lemma NotFreeForm_subst_var_self {s : Nat} (ψ : ArithForm s) (v w : Nat) (h_vw : v ≠
w) :

   NotFreeForm v (substForm v (ArithTerm.var w : ArithTerm s) ψ) := by

 induction ψ with

 | bottom => trivial
```

```
  | eq t1 t2 =>
    dsimp [substForm, NotFreeForm]
    exact ⟨NotOccur_substTerm_self_var t1 v w h_vw, NotOccur_substTerm_self_var t2 v
w h_vw⟩
  | impl f1 f2 ih1 ih2 =>
    dsimp [substForm, NotFreeForm]
    exact ⟨ih1, ih2⟩
  | neg f ih =>
    dsimp [substForm, NotFreeForm]
    exact ih
  | forall_form n f ih =>
    dsimp [substForm]
    by_cases h_vn : v = n
    · rw [if_pos h_vn]
      dsimp [NotFreeForm]
      rw [if_pos h_vn]
      trivial
    · rw [if_neg h_vn]
      dsimp [NotFreeForm]
      rw [if_neg h_vn]
      exact ih
  | provable_op h t =>
    dsimp [substForm, NotFreeForm]
    exact NotOccur_substTerm_self_var t v w h_vw
  | true_op h t =>
    dsimp [substForm, NotFreeForm]
    exact NotOccur_substTerm_self_var t v w h_vw
  | list_nth_eq t1 t2 t3 =>
```

dsimp [substForm, NotFreeForm]

exact ⟨NotOccur_substTerm_self_var t1 v w h_vw, NotOccur_substTerm_self_var t2 v w h_vw, NotOccur_substTerm_self_var t3 v w h_vw⟩

Lemas de cambio de variable ligada (alpha-conversión) y equivalencia en teorías maximales

Se demuestran tres lemas que establecen la validez de la alpha-conversión (renombramiento de variables ligadas) en el sistema SPA. Estos resultados son fundamentales para la manipulación de fórmulas con cuantificadores y son utilizados extensivamente en la construcción de Henkin y en el Teorema de Verdad.

Lema `change_var_direction_1_strong`: Dirección $(\forall w\ \neg \psi[v := \text{var } w]) \to (\forall v\ \neg \psi)$

Establece que a partir de la hipótesis $\forall w\ \neg (\psi[v := \text{var } w])$ se puede deducir $\forall v\ \neg \psi$, bajo las condiciones de que $v \neq w$, w no ocurre en ψ, y v no está ligada en ψ:

$$ \forall v, w \in \mathbb{N},\ \forall \psi \in \text{ArithForm } s,\ v \neq w \land \text{NotOccurForm } w\ \psi \land \text{NoBoundVar } v\ \psi \implies $$

$$ \{\forall w\ \neg (\psi[v := \text{var } w])\} \vdash \forall v\ \neg \psi $$

Estructura de la demostración

1. Se asume $\forall w\ \neg (\psi[v := \text{var } w])$ como hipótesis en el contexto Γ.

2. Se aplica `forall_elim` instanciando w con $\text{var } v$. La condición `IsFreeFor` requerida se verifica usando `IsFreeFor_subst_self`, que requiere precisamente `NotOccurForm w ψ` y `NoBoundVar v ψ`.

3. La fórmula resultante es $\neg (\psi[v := \text{var } w])[w := \text{var } v]$. Por el lema `substForm_substForm_self`, esta doble sustitución colapsa a $\neg \psi$.

4. Se verifica que v no es libre en el contexto (que solo contiene la hipótesis, donde $v \neq w$ garantiza que v no es libre en $\forall w\ \neg (\psi[v := \text{var } w])$ gracias a `NotFreeForm_subst_var_self`).

5. Se aplica `forall_intro` para obtener $\forall v\ \neg \psi$.

Lema `change_var_direction_2_strong`: Dirección $(\forall v\ \neg \psi) \to (\forall w\ \neg \psi[v := \text{var } w])$

Establece la dirección recíproca, bajo condiciones ligeramente más débiles (no se requiere `NoBoundVar v ψ`):

$$ \forall v, w \in \mathbb{N},\ \forall \psi \in \text{ArithForm } s,\ v \neq w \land \text{NotOccurForm } w\ \psi \implies $$

$$ \{\forall v\ \neg \psi\} \vdash \forall w\ \neg (\psi[v := \text{var } w]) $$

La demostración es simétrica: se asume $\forall v\ \neg \psi$, se instancia v con $\text{var } w$ (la condición de libertad se verifica usando `NoBoundVar_of_NotOccurForm` e `IsFreeFor_var`), y se generaliza sobre w (verificando que w no es libre en la hipótesis gracias a $w \neq v$ y `NotFreeForm_of_NotOccurForm`).

Lema `change_var_equivalence_strong`: Equivalencia en teorías maximales

Establece que en una teoría maximal consistente Δ, las fórmulas existenciales con variables ligadas renombradas son equivalentes:

$$ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal } \Delta \land \text{Consistent } \Delta \implies $$

$$ \forall v, w \in \mathbb{N},\ \forall \psi \in \text{ArithForm } s,\ v \neq w \land \text{NotOccurForm } w\ \psi \land \text{NoBoundVar } v\ \psi \implies $$

$$ (\neg \forall v\ \neg \psi) \in \Delta \iff (\neg \forall w\ \neg (\psi[v := \text{var } w])) \in \Delta $$

Estructura de la demostración

Se demuestran dos implicaciones por separado, cada una por contradicción usando maximalidad:

- **Dirección $\Rightarrow$:** Se asume $\neg \forall v\ \neg \psi \in \Delta$ y se supone que $\neg \forall w\ \neg (\psi[v := \text{var } w]) \notin \Delta$. Por maximalidad, $\forall w\ \neg (\psi[v := \text{var } w]) \in \Delta$. Usando `change_var_direction_1_strong` y la clausura deductiva de Δ, se obtiene $\forall v\ \neg \psi \in \Delta$, lo que contradice la consistencia de Δ (pues también contiene su negación).

- **Dirección $\Leftarrow$:** Simétrica, usando `change_var_direction_2_strong`.

Significado

Estos tres lemas constituyen el núcleo de la teoría de alpha-conversión en el sistema SPA. Establecen que el renombramiento de variables ligadas es una operación lícita que preserva la demostrabilidad y la pertenencia a teorías maximales consistentes. En particular, `change_var_equivalence_strong` es el resultado utilizado en la demostración del caso `forall_form` del Teorema de Verdad (a través de `neg_forall_iff_neg_forall_double_neg`) y en la construcción de testigos de Henkin (donde se necesita pasar de una fórmula existencial con una variable ligada a otra equivalente con una variable fresca).

Las condiciones requeridas son naturales y se satisfacen en la construcción de Henkin gracias a la elección de variables frescas: w se elige como una variable completamente nueva (por lo que no ocurre en ψ), y v es la variable original que se desea renombrar (y no está ligada en ψ porque se trabaja con la fórmula existencial donde v es la variable ligada). La conjunción de estos lemas proporciona una justificación completa y rigurosa de la práctica estándar de renombrar variables ligadas para evitar capturas, fundamental en cualquier desarrollo formal de la lógica de primer orden.

```
--
================================================================

-- FUNCIÓN RECURSIVA DE RENOMBRADO DE LIGADURAS (α-CONVERSIÓN)

--
================================================================
```

```
lemma change_var_direction_1_strong {s : Nat} (v w : Nat) (ψ : ArithForm s)

  (h_vw : v ≠ w) (h_w_ψ : NotOccurForm w ψ) (h_bound : NoBoundVar v ψ) :

  SPA_Deduction_Internal s (insert (ArithForm.forall_form w (ArithForm.neg
(substForm v (ArithTerm.var w) ψ))) (fun _ => False))

   (ArithForm.forall_form v (ArithForm.neg ψ)) := by

 let Γ : Set (ArithForm s) := insert (ArithForm.forall_form w (ArithForm.neg
(substForm v (ArithTerm.var w) ψ))) (fun _ => False)

 have h1 : SPA_Deduction_Internal s Γ (ArithForm.forall_form w (ArithForm.neg
(substForm v (ArithTerm.var w) ψ))) :=

  SPA_Deduction_Internal.hyp Γ _ (Or.inl rfl)

 have h_free : IsFreeFor w (ArithTerm.var v) (ArithForm.neg (substForm v
(ArithTerm.var w) ψ)) := by

  rw [IsFreeFor_neg]

  exact IsFreeFor_subst_self ψ v w h_w_ψ h_bound

 have h2 : SPA_Deduction_Internal s Γ (substForm w (ArithTerm.var v)
(ArithForm.neg (substForm v (ArithTerm.var w) ψ))) :=
```

```
  SPA_Deduction_Internal.forall_elim Γ w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)) (ArithTerm.var v) h1 h_free

 have h_eq : substForm w (ArithTerm.var v) (ArithForm.neg (substForm v (ArithTerm.var w) ψ)) = ArithForm.neg ψ := by

  change ArithForm.neg (substForm w (ArithTerm.var v) (substForm v (ArithTerm.var w) ψ)) = ArithForm.neg ψ

  rw [substForm_substForm_self ψ v w h_w_ψ]

 rw [h_eq] at h2

 have h_fresh_v : NotFreeInSet v Γ := by

  intro φ hφ

  rcases hφ with rfl | h_empty

  · dsimp [NotFreeForm]

   rw [if_neg h_vw]

   exact NotFreeForm_subst_var_self ψ v w h_vw

  · nomatch h_empty

 exact SPA_Deduction_Internal.forall_intro Γ v (ArithForm.neg ψ) h2 h_fresh_v


lemma change_var_direction_2_strong {s : Nat} (v w : Nat) (ψ : ArithForm s)

  (h_vw : v ≠ w) (h_w_ψ : NotOccurForm w ψ) :

  SPA_Deduction_Internal s (insert (ArithForm.forall_form v (ArithForm.neg ψ)) (fun _ => False))

   (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ))) := by

 let Γ : Set (ArithForm s) := insert (ArithForm.forall_form v (ArithForm.neg ψ)) (fun _ => False)

 have h1 : SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.neg ψ)) :=

  SPA_Deduction_Internal.hyp Γ _ (Or.inl rfl)

 have h_bound_w : NoBoundVar w ψ := NoBoundVar_of_NotOccurForm w ψ h_w_ψ

 have h_free : IsFreeFor v (ArithTerm.var w) ψ := IsFreeFor_var ψ v w h_bound_w

 have h_elim : SPA_Deduction_Internal s Γ (substForm v (ArithTerm.var w) (ArithForm.neg ψ)) :=
```

```
  SPA_Deduction_Internal.forall_elim Γ v (ArithForm.neg ψ) (ArithTerm.var w) h1 h_free

 rw [substForm_neg] at h_elim

 have h_fresh_w_Γ : NotFreeInSet w Γ := by

  intro φ hφ

  rcases hφ with rfl | h_empty

  · dsimp [NotFreeForm]

   have h_w_not_v : w ≠ v := Ne.symm h_vw

   rw [if_neg h_w_not_v]

   exact NotFreeForm_of_NotOccurForm w ψ h_w_ψ

  · nomatch h_empty

 exact SPA_Deduction_Internal.forall_intro Γ w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)) h_elim h_fresh_w_Γ


lemma change_var_equivalence_strong {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

  (v w : Nat) (ψ : ArithForm s) (h_vw : v ≠ w) (h_w_ψ : NotOccurForm w ψ) (h_bound : NoBoundVar v ψ) :

  (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) ∈ Δ ↔

  (ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)))) ∈ Δ := by

 constructor

 · intro h_in

  by_contra h_not

  have h_not' : ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ))) ∉ Δ := h_not

  have h_forall_w : ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)) ∈ Δ := by

   cases h_max (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ))) with

   | inl h => exact h
```

```
    | inr h => exact absurd h h_not'

  have h_dir1 : SPA_Deduction_Internal s Δ (ArithForm.forall_form v (ArithForm.neg ψ)) := by

    apply SPA_Deduction_Internal.weaken _ _ Δ (change_var_direction_1_strong v w ψ h_vw h_w_ψ h_bound)

    intro x hx; rcases hx with rfl | hx'; exact h_forall_w; nomatch hx'

  have h_forall_v : ArithForm.forall_form v (ArithForm.neg ψ) ∈ Δ :=

    maximal_consistent_closed_under_deduction Δ h_max h_con _ h_dir1

  have h_neg_forall_v : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ)) ∈ Δ := h_in

  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_forall_v) (SPA_Deduction_Internal.hyp Δ _ h_neg_forall_v))

· intro h_in

  by_contra h_not

  have h_not' : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ)) ∉ Δ := h_not

  have h_forall_v : ArithForm.forall_form v (ArithForm.neg ψ) ∈ Δ := by

    cases h_max (ArithForm.forall_form v (ArithForm.neg ψ)) with

    | inl h => exact h

    | inr h => exact absurd h h_not'

  have h_dir2 : SPA_Deduction_Internal s Δ (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ))) := by

    apply SPA_Deduction_Internal.weaken _ _ Δ (change_var_direction_2_strong v w ψ h_vw h_w_ψ)

    intro x hx; rcases hx with rfl | hx'; exact h_forall_v; nomatch hx'

  have h_forall_w : ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)) ∈ Δ :=

    maximal_consistent_closed_under_deduction Δ h_max h_con _ h_dir2

  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_forall_w) (SPA_Deduction_Internal.hyp Δ _ h_in))
```

Definición de `renameBoundVar` y lema de no ligadura

Se define la operación de renombramiento de variables ligadas (alpha-conversión) sobre fórmulas, y se demuestra que esta operación garantiza que la variable original v deja de estar ligada en el resultado.

Función `renameBoundVar`: Renombramiento de variable ligada

Define la operación que reemplaza todas las ocurrencias ligadas de la variable v por la variable u en una fórmula φ:

$$ \text{renameBoundVar}(v, u, \varphi) : \text{ArithForm } s \to \text{ArithForm } s $$

La definición es recursiva estructural:

- **`bottom`, `eq t1 t2`, `provable_op`, `true_op`, `list_nth_eq`:** No contienen cuantificadores, por lo que la fórmula se devuelve inalterada.

- **`impl f1 f2`, `neg f`:** La operación se propaga recursivamente a las subfórmulas.

- **`forall_form n f`:** Si $n = v$ (la variable ligada es la que se desea renombrar), se reemplaza el cuantificador por $\forall u$ y se aplica una sustitución $[v := \text{var } u]$ al cuerpo f (después de renombrar recursivamente en f). Si $n \neq v$, solo se propaga recursivamente al cuerpo.

Intuición

Esta función implementa la alpha-conversión estándar: cambiar el nombre de una variable ligada en todos los cuantificadores donde aparece, ajustando también las ocurrencias ligadas dentro del cuerpo mediante sustitución. La diferencia con una sustitución simple es que solo afecta a las ocurrencias ligadas, no a las libres. La combinación de renombramiento recursivo y sustitución $[v := \text{var } u]$ asegura que todas las referencias a la variable ligada original se actualizan correctamente.

Lema `NoBoundVar_renameBoundVar`: La variable original deja de estar ligada

Establece la propiedad fundamental de la operación de renombramiento: si $v \neq u$, entonces después de renombrar v por u, la variable v no está ligada en el resultado:

$$ \forall \psi \in \text{ArithForm } s,\ \forall v, u \in \mathbb{N},\ v \neq u \implies \text{NoBoundVar } v\ (\text{renameBoundVar } v\ u\ \psi) $$

Estructura de la demostración

Por inducción sobre ψ:

- **Casos sin cuantificadores:** `NoBoundVar` es trivialmente verdadero.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Se analiza si $n = v$:

  - **Si $n = v$:** El cuantificador se renombra a $\forall u$, por lo que la variable ligada pasa a ser u, no v. Se requiere demostrar que v no está ligada en $\forall u\ (\text{substForm } v\ (\text{var } u)\ (\text{renameBoundVar } v\ u\ f))$. Para ello:

1. $v \neq u$ por hipótesis, por lo que v no es la variable ligada del nuevo cuantificador.

2. Por la hipótesis de inducción, `NoBoundVar v (renameBoundVar v u f)`.

3. Por `NoBoundVar_substForm_general_global`, la sustitución $[v := \text{var } u]$ preserva que v no está ligada.

- **Si $n \neq v$:** El cuantificador no se modifica. La condición $v \neq n$ se mantiene, y la hipótesis de inducción garantiza que v no está ligada en el cuerpo.

Significado

`renameBoundVar` y `NoBoundVar_renameBoundVar` proporcionan una implementación constructiva de la alpha-conversión con una garantía verificable: después de renombrar v por u, la variable original v ya no aparece ligada en la fórmula. Esta propiedad es crucial para los lemas de cambio de variable ligada (`change_var_equivalence_strong`), donde una de las hipótesis es precisamente `NoBoundVar v ψ`. Si esta hipótesis no se cumple (porque v está ligada en ψ), se puede aplicar `renameBoundVar` para obtener una fórmula equivalente donde v ya no está ligada, y luego aplicar el lema de cambio de variable.

Esta técnica de "rectificación" de fórmulas es estándar en las demostraciones de completitud: antes de aplicar reglas que requieren que ciertas variables no estén ligadas, se renombran las variables ligadas conflictivas para evitar interferencias. La combinación de `renameBoundVar` (que resuelve el problema de ligaduras) con `change_var_equivalence_strong` (que permite intercambiar variables ligadas preservando la equivalencia) proporciona un mecanismo completo para la manipulación de cuantificadores en el sistema SPA, esencial para la construcción de Henkin y el Teorema de Verdad.

def renameBoundVar {s : Nat} (v u : Nat) (φ : ArithForm s) : ArithForm s :=

```
 match φ with
 | ArithForm.bottom => ArithForm.bottom
 | ArithForm.eq t1 t2 => ArithForm.eq t1 t2
 | ArithForm.impl f1 f2 => ArithForm.impl (renameBoundVar v u f1) (renameBoundVar v u f2)
 | ArithForm.neg f => ArithForm.neg (renameBoundVar v u f)
 | ArithForm.forall_form n f =>
  if n = v then
   ArithForm.forall_form u (substForm v (ArithTerm.var u) (renameBoundVar v u f))
  else
   ArithForm.forall_form n (renameBoundVar v u f)
 | ArithForm.provable_op h t => ArithForm.provable_op h t
 | ArithForm.true_op h t => ArithForm.true_op h t
 | ArithForm.list_nth_eq t1 t2 t3 => ArithForm.list_nth_eq t1 t2 t3

lemma NoBoundVar_renameBoundVar {s : Nat} (ψ : ArithForm s) (v u : Nat) (h_vu : v ≠ u) :
  NoBoundVar v (renameBoundVar v u ψ) := by
 induction ψ generalizing v u with
 | bottom =>
  trivial
 | eq _ _ =>
  trivial
 | impl f1 f2 ih1 ih2 =>
  dsimp [renameBoundVar, NoBoundVar]
  exact ⟨ih1 v u h_vu, ih2 v u h_vu⟩
 | neg f ih =>
  dsimp [renameBoundVar, NoBoundVar]
  exact ih v u h_vu
```

```
 | forall_form n f ih =>
   dsimp [renameBoundVar]
   split_ifs with h_nv
   · subst h_nv
     dsimp [NoBoundVar]
     refine ⟨h_vu, ?_⟩
     apply NoBoundVar_substForm_general_global
     exact ih n u h_vu
   · dsimp [NoBoundVar]
     exact ⟨Ne.symm h_nv, ih v u h_vu⟩
 | provable_op _ _ =>
   trivial
 | true_op _ _ =>
   trivial
 | list_nth_eq _ _ _ =>
   trivial
```

Lemas `renameBoundVar_roundtrip` e `IsFreeFor_subst_self_rename`

Se demuestran dos lemas que establecen propiedades fundamentales de la operación de renombramiento de variables ligadas: la reversibilidad de la operación y la preservación de la condición de libertad.

Lema `renameBoundVar_roundtrip`: Reversibilidad del renombramiento

Establece que el proceso de renombrar v por u y luego aplicar las sustituciones inversas ($u := \text{var } v$ y $v := \text{var } u$) devuelve la fórmula renombrada original:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, u \in \mathbb{N},\ v \neq u \land \text{NotOccurForm } u\ \varphi \implies $$

$$ \text{substForm } u\ (\text{var } v)\ (\text{substForm } v\ (\text{var } u)\ (\text{renameBoundVar } v\ u\ \varphi)) = \text{renameBoundVar } v\ u\ \varphi $$

Estructura de la demostración

Por inducción sobre φ:

- **Casos sin cuantificadores:** Se aplica `substTerm_substTerm_self` a los subtérminos, que establece que dos sustituciones intercambiadas sobre variables frescas se cancelan.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form n f`:** Se distingue si $n = v$:

  - **Si $n = v$:** El cuantificador ha sido renombrado a $\forall u$. La primera sustitución ($v := \text{var } u$) no penetra el cuantificador $\forall u$ porque la variable ligada es u, no v. La segunda sustitución ($u := \text{var } v$) atraviesa el cuantificador (pues $u \neq v$). Se utiliza `substForm_substForm_same` para colapsar las sustituciones dentro del cuerpo.

  - **Si $n \neq v$:** El cuantificador no ha sido modificado. Se verifica que $u \neq n$ (por `NotOccurForm u (forall_form n f)`) y $v \neq n$, por lo que ambas sustituciones atraviesan el cuantificador. Se aplica la hipótesis de inducción al cuerpo.

Significado

Este lema muestra que `renameBoundVar` es una operación reversible en el sentido de que las sustituciones que la acompañan (al aplicar la alpha-conversión) pueden deshacerse. Es una propiedad de coherencia que garantiza que el renombramiento no altera el significado de la fórmula de manera irreversible. La condición `NotOccurForm u φ` asegura que la nueva variable u es fresca respecto a la fórmula original, lo cual es la hipótesis estándar para que el renombramiento sea seguro.

Lema `IsFreeFor_subst_self_rename`: Preservación de la libertad bajo renombramiento

Establece que después de renombrar n por u y aplicar la sustitución correspondiente, la variable n es libre para u en la fórmula resultante:

$$ \forall f \in \text{ArithForm } s,\ \forall n, u \in \mathbb{N},\ n \neq u \land \text{NotOccurForm } u\ f \implies $$

$$ \text{IsFreeFor } u\ (\text{var } n)\ (\text{substForm } n\ (\text{var } u)\ (\text{renameBoundVar } n\ u\ f)) $$

Estructura de la demostración

Por inducción sobre f:

- **Casos sin cuantificadores:** `IsFreeFor` es trivialmente verdadero.

- **`impl f1 f2`, `neg f`:** Se aplican las hipótesis de inducción.

- **`forall_form m f`:** Se distingue si $m = n$:

  - **Si $m = n$:** El cuantificador ha sido renombrado a $\forall u$. La sustitución $n := \text{var } u$ no penetra el cuantificador $\forall u$. La condición `IsFreeFor u (var

n) (forall_form u ...)` se evalúa: como $u = u$, la sustitución no penetra y la condición es trivial.

- **Si $m \neq n$:** El cuantificador no ha sido modificado. La sustitución $n := \text{var } u$ atraviesa el cuantificador (pues $m \neq n$). Se requiere verificar `IsFreeFor u (var n) (forall_form m ...)`. Como $u \neq m$ (por `NotOccurForm u (forall_form m f)`), se necesita `NotOccur m (var n)` (cierto porque $m \neq n$) e `IsFreeFor u (var n)` en el cuerpo, que se obtiene por hipótesis de inducción.

Significado

Este lema es técnicamente importante para la aplicación de `forall_elim` en contextos donde se ha realizado alpha-conversión. Cuando se renombra una variable ligada n a u y luego se necesita instanciar el cuantificador resultante, la condición `IsFreeFor` debe verificarse. El lema garantiza que la variable original n es libre para u en la fórmula transformada, lo cual es un paso intermedio necesario en las demostraciones de cambio de variable ligada.

Junto con `renameBoundVar_roundtrip`, estos lemas proporcionan una base sólida para la manipulación de fórmulas con cuantificadores en el sistema SPA. La operación `renameBoundVar` y sus propiedades son utilizadas en la rectificación de fórmulas (asegurar que ciertas variables no están ligadas) antes de aplicar reglas de inferencia que requieren condiciones de frescura, como `forall_intro` y los lemas de cambio de variable. Constituyen la implementación formal de la práctica estándar en lógica de "renombrar variables ligadas para evitar conflictos".

```
lemma renameBoundVar_roundtrip {s : Nat} (φ : ArithForm s) (v u : Nat)
  (h_vu : v ≠ u) (h_u_φ : NotOccurForm u φ) :
  substForm u (ArithTerm.var v) (substForm v (ArithTerm.var u) (renameBoundVar v u φ))
    = renameBoundVar v u φ := by
 induction φ generalizing v u h_vu with
```

```
  | bottom => rfl
  | eq t1 t2 =>
    simp only [renameBoundVar, substForm_eq]
    rw [substTerm_substTerm_self t1 v u h_u_φ.1,
      substTerm_substTerm_self t2 v u h_u_φ.2]
  | impl f1 f2 ih1 ih2 =>
    simp only [renameBoundVar, substForm_impl]
    rw [ih1 v u h_vu h_u_φ.1, ih2 v u h_vu h_u_φ.2]
  | neg f ih =>
    simp only [renameBoundVar, substForm_neg]
    rw [ih v u h_vu h_u_φ]
  | forall_form n f ih =>
    by_cases h_nv : n = v
    · subst h_nv
      rw [renameBoundVar]
      rw [if_pos rfl]
      rw [substForm_forall_form]
      rw [if_neg h_vu]
      rw [substForm_forall_form]
      rw [if_pos rfl]
      congr 1
      rw [← substForm_substForm_same]
      have h_subst_term : substTerm n (ArithTerm.var u : ArithTerm s) (ArithTerm.var u
: ArithTerm s) = (ArithTerm.var u : ArithTerm s) := by
        unfold substTerm
        rw [if_neg h_vu]
      rw [h_subst_term]
    · have h_un : u ≠ n := h_u_φ.1
```

```
    have h_vn : v ≠ n := Ne.symm h_nv
    rw [renameBoundVar]
    rw [if_neg h_nv]
    rw [substForm_forall_form, if_neg h_vn]
    rw [substForm_forall_form, if_neg h_un]
    congr 1
    exact ih v u h_vu h_u_φ.2
  | provable_op heq t =>
   simp only [renameBoundVar, substForm_provable_op]
   rw [substTerm_substTerm_self t v u h_u_φ]
  | true_op heq t =>
   simp only [renameBoundVar, substForm_true_op]
   rw [substTerm_substTerm_self t v u h_u_φ]
  | list_nth_eq t1 t2 t3 =>
   simp only [renameBoundVar, substForm_list_nth_eq]
   rw [substTerm_substTerm_self t1 v u h_u_φ.1,
      substTerm_substTerm_self t2 v u h_u_φ.2.1,
      substTerm_substTerm_self t3 v u h_u_φ.2.2]
```

```
lemma IsFreeFor_subst_self_rename {s : Nat} (f : ArithForm s) (n u : Nat) (h_vu : n ≠ u)
(h_u_f : NotOccurForm u f) :
    IsFreeFor u (ArithTerm.var n) (substForm n (ArithTerm.var u) (renameBoundVar n
u f)) := by
  induction f generalizing n u with
```

```
| bottom => trivial
| eq t1 t2 => trivial
| impl f1 f2 ih1 ih2 =>
  dsimp [renameBoundVar, substForm, IsFreeFor]
  exact ⟨ih1 n u h_vu h_u_f.1, ih2 n u h_vu h_u_f.2⟩
| neg f ih =>
  dsimp [renameBoundVar, substForm, IsFreeFor]
  exact ih n u h_vu h_u_f
| forall_form m f ih =>
  dsimp [renameBoundVar]
  by_cases h_mn : m = n
  · subst h_mn
    rw [if_pos rfl]
    dsimp [substForm]
    rw [if_neg h_vu]
    dsimp [IsFreeFor]
    rw [if_pos rfl]
    trivial
  · rw [if_neg h_mn]
    rw [substForm_forall_form]
    rw [if_neg (Ne.symm h_mn)]
    dsimp [IsFreeFor]
    rw [if_neg h_u_f.1]
    refine ⟨NotOccur.var n h_mn, ?_⟩
    exact ih n u h_vu h_u_f.2
| provable_op _ _ => trivial
| true_op _ _ => trivial
| list_nth_eq _ _ _ => trivial
```

Lema `rename_equiv`: Equivalencia deductiva bajo renombramiento de variables ligadas

Se demuestra un resultado fundamental que establece que el renombramiento de variables ligadas preserva la equivalencia deductiva en contexto vacío: una fórmula con una variable ligada renombrada es deductivamente equivalente a la fórmula original, incluso después de aplicar una sustitución sobre la variable original.

Enunciado formal

$$ \forall v, u \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \forall \psi \in \text{ArithForm } s,\ v \neq u \land \text{NotOccurForm } u\ \psi \implies $$

$$ \emptyset \vdash (\psi_{\text{ren}}[v := t]) \to (\psi[v := t]) \quad \land \quad \emptyset \vdash (\psi[v := t]) \to (\psi_{\text{ren}}[v := t]) $$

donde $\psi_{\text{ren}} = \text{renameBoundVar } v\ u\ \psi$.

Estructura de la demostración

La prueba procede por inducción generalizada sobre ψ, demostrando simultáneamente ambas direcciones para cada constructor. Se utiliza una inducción interna que recorre todos los casos:

- **Casos base (`bottom`, `eq`, `provable_op`, `true_op`, `list_nth_eq`):** `renameBoundVar` no modifica estas fórmulas, por lo que la equivalencia es trivial (identidad).

- **`impl f1 f2`:** Se utilizan las hipótesis de inducción para f_1 y f_2. La demostración es un ejercicio de lógica proposicional: a partir de las equivalencias para las subfórmulas, se construyen las implicaciones para la implicación compuesta usando `imp_intro`, `imp_elim` y manipulaciones del contexto.

- **`neg f`:** Similar, usando las equivalencias para f y las reglas de negación (`neg_intro`, `neg_elim`).

- **`forall_form n f`:** Es el caso más complejo. Se distingue si $n = v$ (la variable ligada es la que se renombra) o no:

  - **Si $n = v$:** La fórmula original es $\forall v\ f$ y la renombrada es $\forall u\ (f_{\text{ren}}[v := \text{var } u])$. Para cada dirección, se instancia el cuantificador con una variable adecuada, se aplica la hipótesis de inducción (con $t := \text{var } u$ o $t := \text{var } n$), y se utilizan las reglas de generalización (`forall_intro`) verificando las condiciones de no libertad. Se emplean también `substForm_self_var` y `substForm_substForm_same` para simplificar las sustituciones.

  - **Si $n \neq v$:** El cuantificador no se modifica. Ambas direcciones se demuestran instanciando el cuantificador con $\text{var } n$, aplicando la hipótesis de inducción para el cuerpo f, y generalizando de nuevo. Las condiciones de no libertad se verifican porque la variable ligada n no es libre en las hipótesis (por la condición `NotFreeForm` evaluada en `if_pos rfl`).

Significado

`rename_equiv` es un resultado de gran alcance que establece la legitimidad de la alpha-conversión en el sistema SPA. No solo demuestra que dos fórmulas que difieren solo en el nombre de una variable ligada son equivalentes (caso particular con $t = \text{var } v$), sino que la equivalencia se preserva bajo cualquier sustitución $[v := t]$ sobre la variable original. Esto es crucial para las aplicaciones en el Teorema de Verdad y la construcción de Henkin, donde las fórmulas se manipulan bajo sustituciones que involucran términos cerrados.

La demostración, aunque extensa, sigue un patrón sistemático: para cada constructor lógico, se utiliza la hipótesis de inducción sobre las subfórmulas y se aplican las reglas de inferencia correspondientes. El caso de `forall_form` es el más elaborado porque requiere coordinar la interacción entre el renombramiento, las sustituciones, y las condiciones de frescura para la generalización universal.

Este lema es la justificación formal de la práctica común en lógica de "renombrar variables ligadas para evitar conflictos": el significado de una fórmula no cambia si se renombra una variable ligada por una variable fresca, incluso cuando la fórmula aparece como parte de una sustitución más amplia. La condición `NotOccurForm u ψ` (la nueva variable no ocurre en la fórmula original) es la hipótesis estándar de frescura que garantiza que el renombramiento es seguro y reversible. Sin esta condición, el renombramiento podría causar captura de variables libres, alterando el significado de la fórmula.

lemma rename_equiv {s : Nat} (v u : Nat) (t : ArithTerm s) (ψ : ArithForm s) (h_vu : v ≠ u) (h_u_ψ : NotOccurForm u ψ) :

SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl (substForm v t (renameBoundVar v u ψ)) (substForm v t ψ)) ∧

```
  SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl (substForm v t ψ) (substForm v t (renameBoundVar v u ψ))) := by

 have h_all : ∀ (ψ : ArithForm s) (h_u_ψ : NotOccurForm u ψ) (t : ArithTerm s),

   SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl (substForm v t (renameBoundVar v u ψ)) (substForm v t ψ)) ∧

   SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl (substForm v t ψ) (substForm v t (renameBoundVar v u ψ))) := by

  intro ψ

  induction ψ with

  | bottom =>

   intro h_u_ψ t

   constructor

   · exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

   · exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

  | eq t1 t2 =>

   intro h_u_ψ t

   constructor

   · exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

   · exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

  | impl f1 f2 ih1 ih2 =>

   intro h_u_ψ t

   have h1 := ih1 h_u_ψ.1 t

   have h2 := ih2 h_u_ψ.2 t

   constructor

   · apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro
```

have h_BD : SPA_Deduction_Internal s (insert (substForm v t f1) (insert (substForm v t (renameBoundVar v u (ArithForm.impl f1 f2))) ∅)) (ArithForm.impl (substForm v t (renameBoundVar v u f2)) (substForm v t f2)) :=

SPA_Deduction_Internal.weaken ∅ _ _ h2.left (by intro x hx; nomatch hx)

have h_B1A1 : SPA_Deduction_Internal s (insert (substForm v t f1) (insert (substForm v t (renameBoundVar v u (ArithForm.impl f1 f2))) ∅)) (ArithForm.impl (substForm v t f1) (substForm v t (renameBoundVar v u f1))) :=

SPA_Deduction_Internal.weaken ∅ _ _ h1.right (by intro x hx; nomatch hx)

have h_C : SPA_Deduction_Internal s (insert (substForm v t f1) (insert (substForm v t (renameBoundVar v u (ArithForm.impl f1 f2))) ∅)) (substForm v t f1) :=

SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)

have h_A : SPA_Deduction_Internal s (insert (substForm v t f1) (insert (substForm v t (renameBoundVar v u (ArithForm.impl f1 f2))) ∅)) (substForm v t (renameBoundVar v u f1)) :=

SPA_Deduction_Internal.imp_elim _ _ _ h_B1A1 h_C

have h_AB : SPA_Deduction_Internal s (insert (substForm v t f1) (insert (substForm v t (renameBoundVar v u (ArithForm.impl f1 f2))) ∅)) (substForm v t (renameBoundVar v u (ArithForm.impl f1 f2))) :=

SPA_Deduction_Internal.hyp _ _ (Or.inr (Or.inl rfl))

have h_B : SPA_Deduction_Internal s (insert (substForm v t f1) (insert (substForm v t (renameBoundVar v u (ArithForm.impl f1 f2))) ∅)) (substForm v t (renameBoundVar v u f2)) :=

SPA_Deduction_Internal.imp_elim _ _ _ h_AB h_A

exact SPA_Deduction_Internal.imp_elim _ _ _ h_BD h_B

· apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

have h_BD : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar v u f1)) (insert (substForm v t (ArithForm.impl f1 f2)) ∅)) (ArithForm.impl (substForm v t f2) (substForm v t (renameBoundVar v u f2))) :=

SPA_Deduction_Internal.weaken ∅ _ _ h2.right (by intro x hx; nomatch hx)

have h_B1A1 : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar v u f1)) (insert (substForm v t (ArithForm.impl f1 f2)) ∅)) (ArithForm.impl (substForm v t (renameBoundVar v u f1)) (substForm v t f1)) :=

SPA_Deduction_Internal.weaken ∅ _ _ h1.left (by intro x hx; nomatch hx)

```
      have h_C : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar v u
f1)) (insert (substForm v t (ArithForm.impl f1 f2)) ∅)) (substForm v t
(renameBoundVar v u f1)) :=

        SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)

      have h_A : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar v u
f1)) (insert (substForm v t (ArithForm.impl f1 f2)) ∅)) (substForm v t f1) :=

        SPA_Deduction_Internal.imp_elim _ _ _ h_B1A1 h_C

      have h_AB : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar v
u f1)) (insert (substForm v t (ArithForm.impl f1 f2)) ∅)) (substForm v t
(ArithForm.impl f1 f2)) :=

        SPA_Deduction_Internal.hyp _ _ (Or.inr (Or.inl rfl))

      have h_B : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar v u
f1)) (insert (substForm v t (ArithForm.impl f1 f2)) ∅)) (substForm v t f2) :=

        SPA_Deduction_Internal.imp_elim _ _ _ h_AB h_A

      exact SPA_Deduction_Internal.imp_elim _ _ _ h_BD h_B

    | neg f ih =>

      intro h_u_ψ t

      have h_f := ih h_u_ψ t

      constructor

      · apply SPA_Deduction_Internal.imp_intro

        apply SPA_Deduction_Internal.neg_intro

      have h_BA : SPA_Deduction_Internal s (insert (substForm v t f) (insert
(ArithForm.neg (substForm v t (renameBoundVar v u f))) ∅)) (ArithForm.impl
(substForm v t f) (substForm v t (renameBoundVar v u f))) :=

        SPA_Deduction_Internal.weaken ∅ _ _ h_f.right (by intro x hx; nomatch hx)

      have h_B : SPA_Deduction_Internal s (insert (substForm v t f) (insert
(ArithForm.neg (substForm v t (renameBoundVar v u f))) ∅)) (substForm v t f) :=

        SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)

      have h_A : SPA_Deduction_Internal s (insert (substForm v t f) (insert
(ArithForm.neg (substForm v t (renameBoundVar v u f))) ∅)) (substForm v t
(renameBoundVar v u f)) :=

        SPA_Deduction_Internal.imp_elim _ _ _ h_BA h_B
```

```
      have h_negA : SPA_Deduction_Internal s (insert (substForm v t f) (insert
(ArithForm.neg (substForm v t (renameBoundVar v u f))) ∅)) (ArithForm.neg
(substForm v t (renameBoundVar v u f))) :=

        SPA_Deduction_Internal.hyp _ _ (Or.inr (Or.inl rfl))

      exact SPA_Deduction_Internal.neg_elim _ _ h_A h_negA

    · apply SPA_Deduction_Internal.imp_intro

      apply SPA_Deduction_Internal.neg_intro

      have h_BA : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar v
u f)) (insert (ArithForm.neg (substForm v t f)) ∅)) (ArithForm.impl (substForm v t
(renameBoundVar v u f)) (substForm v t f)) :=

        SPA_Deduction_Internal.weaken ∅ _ _ h_f.left (by intro x hx; nomatch hx)

      have h_B : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar v u
f)) (insert (ArithForm.neg (substForm v t f)) ∅)) (substForm v t (renameBoundVar v u
f)) :=

        SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)

      have h_A : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar v u
f)) (insert (ArithForm.neg (substForm v t f)) ∅)) (substForm v t f) :=

        SPA_Deduction_Internal.imp_elim _ _ _ h_BA h_B

      have h_negA : SPA_Deduction_Internal s (insert (substForm v t (renameBoundVar
v u f)) (insert (ArithForm.neg (substForm v t f)) ∅)) (ArithForm.neg (substForm v t f))
:=

        SPA_Deduction_Internal.hyp _ _ (Or.inr (Or.inl rfl))

      exact SPA_Deduction_Internal.neg_elim _ _ h_A h_negA

  | forall_form n f ih =>

    intro h_u_ψ t

    by_cases h_nv : n = v

    · subst h_nv

      have h_u_f : NotOccurForm u f := h_u_ψ.2

      have h_ih_u := ih h_u_f (ArithTerm.var u)

      have h_ih_left := h_ih_u.left

      have h_ih_right := h_ih_u.right
```

```
    have h_LHS_eq : substForm n t (renameBoundVar n u (ArithForm.forall_form n f))
=
            ArithForm.forall_form u (substForm n (ArithTerm.var u)
(renameBoundVar n u f)) := by
      rw [renameBoundVar]
      rw [if_pos rfl]
      rw [substForm_forall_form]
      rw [if_neg h_vu]
      congr 1
      rw [← substForm_substForm_same]
      have h_subst_term : substTerm n t (ArithTerm.var u : ArithTerm s) =
(ArithTerm.var u : ArithTerm s) := by
       unfold substTerm
       rw [if_neg h_vu]
      rw [h_subst_term]
    have h_RHS_eq : substForm n t (ArithForm.forall_form n f) =
ArithForm.forall_form n f := by
      unfold substForm
      rw [if_pos rfl]
    rw [h_LHS_eq, h_RHS_eq]
    constructor
    · apply SPA_Deduction_Internal.imp_intro
      let Γ := insert (ArithForm.forall_form u (substForm n (ArithTerm.var u)
(renameBoundVar n u f))) (∅ : Set (ArithForm s))
      have h_hyp : SPA_Deduction_Internal s Γ (ArithForm.forall_form u (substForm n
(ArithTerm.var u) (renameBoundVar n u f))) :=
       SPA_Deduction_Internal.hyp Γ _ (Or.inl rfl)
      have h_elim : SPA_Deduction_Internal s Γ (substForm u (ArithTerm.var n)
(substForm n (ArithTerm.var u) (renameBoundVar n u f))) := by
       apply SPA_Deduction_Internal.forall_elim Γ u (substForm n (ArithTerm.var u)
(renameBoundVar n u f)) (ArithTerm.var n) h_hyp
```

```
      exact IsFreeFor_subst_self_rename f n u h_vu h_u_f
    rw [renameBoundVar_roundtrip f n u h_vu h_u_f] at h_elim
    have h_iff0 := ih h_u_f (ArithTerm.var n)
    rw [substForm_self_var, substForm_self_var] at h_iff0
    have h_g_to_f : SPA_Deduction_Internal s Γ f := by
      have h_g_to_f' : SPA_Deduction_Internal s Γ (ArithForm.impl (renameBoundVar n u f) f) :=
        SPA_Deduction_Internal.weaken (∅ : Set (ArithForm s)) _ Γ h_iff0.left (Set.empty_subset _)
      exact SPA_Deduction_Internal.imp_elim _ _ _ h_g_to_f' h_elim
    have h_fresh_n : NotFreeInSet n Γ := by
      intro φ hφ
      rcases hφ with rfl | h_empty
      · dsimp [NotFreeForm]
        rw [if_neg h_vu]
        exact NotFreeForm_subst_var_self_global (renameBoundVar n u f) n u h_vu
      · nomatch h_empty
    exact SPA_Deduction_Internal.forall_intro Γ n f h_g_to_f h_fresh_n
  · apply SPA_Deduction_Internal.imp_intro
    let Γ := insert (ArithForm.forall_form n f) (∅ : Set (ArithForm s))
    have h_hyp : SPA_Deduction_Internal s Γ (ArithForm.forall_form n f) :=
      SPA_Deduction_Internal.hyp Γ _ (Or.inl rfl)
    have h_elim : SPA_Deduction_Internal s Γ (substForm n (ArithTerm.var u) f) := by
      apply SPA_Deduction_Internal.forall_elim Γ n f (ArithTerm.var u) h_hyp
      exact IsFreeFor_var f n u (NoBoundVar_of_NotOccurForm u f h_u_f)
    have h_ih_weak : SPA_Deduction_Internal s Γ (ArithForm.impl (substForm n (ArithTerm.var u) f) (substForm n (ArithTerm.var u) (renameBoundVar n u f))) := by
      apply SPA_Deduction_Internal.weaken (∅ : Set (ArithForm s)) _ Γ h_ih_right (Set.empty_subset _)
```

```
      have h_A : SPA_Deduction_Internal s Γ (substForm n (ArithTerm.var u) (renameBoundVar n u f)) :=
        SPA_Deduction_Internal.imp_elim Γ _ _ h_ih_weak h_elim
      have h_fresh_u : NotFreeInSet u Γ := by
        intro φ hφ
        rcases hφ with rfl | h_empty
        · dsimp [NotFreeForm]
          rw [if_neg h_vu.symm]
          exact NotFreeForm_of_NotOccurForm u f h_u_f
        · nomatch h_empty
      exact SPA_Deduction_Internal.forall_intro Γ u (substForm n (ArithTerm.var u) (renameBoundVar n u f)) h_A h_fresh_u
    · constructor
      · have h_vn : v ≠ n := Ne.symm h_nv
        rw [renameBoundVar]
        rw [if_neg h_nv]
        rw [substForm_forall_form, if_neg h_vn]
        rw [substForm_forall_form, if_neg h_vn]
        apply SPA_Deduction_Internal.imp_intro
        apply SPA_Deduction_Internal.forall_intro
        · have h_elim : SPA_Deduction_Internal s (insert (ArithForm.forall_form n (substForm v t (renameBoundVar v u f))) (∅ : Set (ArithForm s))) (substForm n (ArithTerm.var n) (substForm v t (renameBoundVar v u f))) := by
            apply SPA_Deduction_Internal.forall_elim (insert (ArithForm.forall_form n (substForm v t (renameBoundVar v u f))) (∅ : Set (ArithForm s))) n (substForm v t (renameBoundVar v u f)) (ArithTerm.var n)
            · apply SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
            · exact IsFreeFor_var_self _ n
          rw [substForm_self_var] at h_elim
          have h_f := ih h_u_ψ.2 t
```

have h_ih_weak : SPA_Deduction_Internal s (insert (ArithForm.forall_form n (substForm v t (renameBoundVar v u f))) (∅ : Set (ArithForm s))) (ArithForm.impl (substForm v t (renameBoundVar v u f)) (substForm v t f)) := by

apply SPA_Deduction_Internal.weaken (∅ : Set (ArithForm s)) _ _ h_f.left (by intro x hx; nomatch hx)

exact SPA_Deduction_Internal.imp_elim _ _ _ h_ih_weak h_elim

· intro φ hφ

rcases hφ with h_eq_φ | h_empty

· subst h_eq_φ

dsimp [NotFreeForm]

rw [if_pos rfl]

trivial

· nomatch h_empty

· have h_vn : v ≠ n := Ne.symm h_nv

rw [renameBoundVar]

rw [if_neg h_nv]

rw [substForm_forall_form, if_neg h_vn]

rw [substForm_forall_form, if_neg h_vn]

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.forall_intro

· have h_elim : SPA_Deduction_Internal s (insert (ArithForm.forall_form n (substForm v t f)) (∅ : Set (ArithForm s))) (substForm n (ArithTerm.var n) (substForm v t f)) := by

apply SPA_Deduction_Internal.forall_elim (insert (ArithForm.forall_form n (substForm v t f)) (∅ : Set (ArithForm s))) n (substForm v t f) (ArithTerm.var n)

· apply SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)

· exact IsFreeFor_var_self _ n

rw [substForm_self_var] at h_elim

have h_f := ih h_u_ψ.2 t

```
        have h_ih_weak : SPA_Deduction_Internal s (insert (ArithForm.forall_form n (substForm v t f)) (∅ : Set (ArithForm s))) (ArithForm.impl (substForm v t f) (substForm v t (renameBoundVar v u f))) := by

          apply SPA_Deduction_Internal.weaken (∅ : Set (ArithForm s)) _ _ h_f.right (by intro x hx; nomatch hx)

        exact SPA_Deduction_Internal.imp_elim _ _ _ h_ih_weak h_elim

      · intro φ hφ

        rcases hφ with h_eq_φ | h_empty

        · subst h_eq_φ

          dsimp [NotFreeForm]

          rw [if_pos rfl]

          trivial

        · nomatch h_empty

  | provable_op h t' =>

    intro h_u_ψ t

    constructor

    · exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

    · exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

  | true_op h t' =>

    intro h_u_ψ t

    constructor

    · exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

    · exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

  | list_nth_eq t1 t2 t3 =>

    intro h_u_ψ t

    constructor
```

· exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

· exact SPA_Deduction_Internal.imp_intro _ _ _ (SPA_Deduction_Internal.hyp _ _ (Or.inl rfl))

exact h_all ψ h_u_ψ t

Lema `alpha_rename_inside`: Preservación de la equivalencia existencial bajo renombramiento

Se demuestra un lema que establece que el renombramiento de una variable ligada dentro de una fórmula existencial preserva la implicación deductiva: si se satisface $\exists v\ \psi$, entonces también se satisface $\exists v\ \psi_{\text{ren}}$, donde ψ_{ren} es la fórmula con la variable ligada renombrada.

Enunciado formal

$$ \forall v, u \in \mathbb{N},\ \forall \psi \in \text{ArithForm } s,\ v \neq u \land \text{NotOccurForm } u\ \psi \implies $$

$$ \emptyset \vdash (\neg \forall v\ \neg \psi) \to (\neg \forall v\ \neg (\text{renameBoundVar } v\ u\ \psi)) $$

Estructura de la demostración

1. **Configuración del contexto:** Se asume $\neg \forall v\ \neg \psi$ y, por contradicción, se supone $\forall v\ \neg (\text{renameBoundVar } v\ u\ \psi)$. El objetivo es derivar $\bot$ en este contexto extendido.

2. **Instanciación del cuantificador:** De $\forall v\ \neg (\text{renameBoundVar } v\ u\ \psi)$, se elimina el cuantificador universal instanciando v con $\text{var } v$ (usando `IsFreeFor_var_self`). Se obtiene $\neg ((\text{renameBoundVar } v\ u\ \psi)[v := \text{var } v])$, que por `substForm_self_var` es simplemente $\neg (\text{renameBoundVar } v\ u\ \psi)$.

3. **Uso de la equivalencia `rename_equiv`:** El lema `rename_equiv` (con $t := \text{var } v$) proporciona $\emptyset \vdash (\text{renameBoundVar } v\ u\ \psi) \to \psi$ y $\emptyset \vdash \psi \to (\text{renameBoundVar } v\ u\ \psi)$. Estas implicaciones se debilitan al contexto extendido.

4. **Derivación de $\neg \psi$:** A partir de $\neg (\text{renameBoundVar } v\ u\ \psi)$ y la implicación $\psi \to (\text{renameBoundVar } v\ u\ \psi)$, se deduce $\neg \psi$ por `neg_intro` (si ψ fuera cierto, se obtendría el renombrado, contradiciendo su negación).

5. **Generalización:** Dado que v no es libre en el contexto (el contexto solo contiene las dos hipótesis, y en ambas v está ligada por los cuantificadores), se aplica `forall_intro` para obtener $\forall v\ \neg \psi$.

6. **Contradicción:** El contexto también contiene $\neg \forall v\ \neg \psi$ (la hipótesis original). La presencia simultánea de $\forall v\ \neg \psi$ y $\neg \forall v\ \neg \psi$ produce $\bot$ por `neg_elim`.

7. **Descarga:** Por `neg_intro`, se descarga la hipótesis $\forall v\ \neg (\text{renameBoundVar } v\ u\ \psi)$ para obtener su negación. Por `imp_intro`, se obtiene la implicación buscada.

Significado

`alpha_rename_inside` es el resultado que permite "rectificar" fórmulas existenciales antes de aplicar la propiedad de Henkin. En la construcción de Henkin, cuando se procesa una fórmula existencial $\exists v\ \psi$, puede ocurrir que v esté ligada en ψ (es decir, aparezca en un cuantificador interno), lo que impide aplicar directamente los lemas de cambio de variable. En tal caso, se puede usar `alpha_rename_inside` para reemplazar $\exists v\ \psi$ por $\exists v\ (\text{renameBoundVar } v\ u\ \psi)$, donde u es una variable fresca. Como esta nueva fórmula es implicada por la original, si la original pertenece a Δ, la nueva también pertenece (por clausura deductiva). Además, en la nueva fórmula v ya no está ligada en el cuerpo (por `NoBoundVar_renameBoundVar`), por lo que se pueden aplicar los lemas de cambio de variable.

Este lema, junto con `rename_equiv` y `NoBoundVar_renameBoundVar`, completa la maquinaria de rectificación de fórmulas: es posible transformar cualquier fórmula existencial en una equivalente donde la variable ligada no aparece ligada en el cuerpo, lo cual es un prerrequisito para aplicar la propiedad de Henkin. La demostración utiliza de manera elegante las equivalencias establecidas por `rename_equiv` (para $t = \text{var } v$, es decir, sin sustitución adicional) y las reglas estándar de manipulación de cuantificadores y negaciones. La condición $v \neq u$ y `NotOccurForm u ψ` garantizan que el renombramiento es seguro y no introduce capturas.

lemma alpha_rename_inside {s : Nat} (v u : Nat) (ψ : ArithForm s) (h_vu : v ≠ u) (h_u_ψ : NotOccurForm u ψ) :

  SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ)))

```
    (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ))))) := by

  apply SPA_Deduction_Internal.imp_intro

  apply SPA_Deduction_Internal.neg_intro

  let Γ := insert (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ))) (insert (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) (∅ : Set (ArithForm s)))

  have h_forall : SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ))) :=

    SPA_Deduction_Internal.hyp Γ _ (Or.inl rfl)

  have h_elim : SPA_Deduction_Internal s Γ (ArithForm.neg (substForm v (ArithTerm.var v) (renameBoundVar v u ψ))) := by

    have h_elim_raw := SPA_Deduction_Internal.forall_elim Γ v (ArithForm.neg (renameBoundVar v u ψ)) (ArithTerm.var v) h_forall (IsFreeFor_var_self _ v)

    rw [substForm_neg] at h_elim_raw

    exact h_elim_raw

  have h_eq : substForm v (ArithTerm.var v) ψ = ψ := substForm_self_var ψ v

  have h_eq' : substForm v (ArithTerm.var v) (renameBoundVar v u ψ) = renameBoundVar v u ψ :=

    substForm_self_var (renameBoundVar v u ψ) v

  rw [h_eq'] at h_elim

  have h_equiv := rename_equiv v u (ArithTerm.var v) ψ h_vu h_u_ψ

  have h_equiv_subst : SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl (renameBoundVar v u ψ) ψ) := by

    have h_equiv_raw := h_equiv.left

    rw [substForm_self_var (renameBoundVar v u ψ) v, substForm_self_var ψ v] at h_equiv_raw

    exact h_equiv_raw

  have h_equiv_subst' : SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl ψ (renameBoundVar v u ψ)) := by

    have h_equiv_raw := h_equiv.right
```

```
  rw [substForm_self_var ψ v, substForm_self_var (renameBoundVar v u ψ) v] at h_equiv_raw
  exact h_equiv_raw
 have h_neg_ψ : SPA_Deduction_Internal s Γ (ArithForm.neg ψ) := by
  apply SPA_Deduction_Internal.neg_intro
  have h_ψ : SPA_Deduction_Internal s (insert ψ Γ) ψ := SPA_Deduction_Internal.hyp _ ψ (Or.inl rfl)
  have h_neg_ψ'_weak : SPA_Deduction_Internal s (insert ψ Γ) (ArithForm.neg (renameBoundVar v u ψ)) :=
    SPA_Deduction_Internal.weaken Γ _ _ h_elim (Set.subset_insert _ _)
  have h_ψ' : SPA_Deduction_Internal s (insert ψ Γ) (renameBoundVar v u ψ) := by
   have h_equiv_right_weak : SPA_Deduction_Internal s (insert ψ Γ) (ArithForm.impl ψ (renameBoundVar v u ψ)) := by
     apply SPA_Deduction_Internal.weaken ∅ _ _ h_equiv_subst' (by intro x hx; nomatch hx)
    exact SPA_Deduction_Internal.imp_elim _ ψ _ h_equiv_right_weak h_ψ
  exact SPA_Deduction_Internal.neg_elim _ _ h_ψ' h_neg_ψ'_weak
 have h_forall_v : SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.neg ψ)) := by
  apply SPA_Deduction_Internal.forall_intro _ v (ArithForm.neg ψ) h_neg_ψ
  intro φ hφ
  rcases hφ with h_eq_φ | h_in
  · subst h_eq_φ
   dsimp [NotFreeForm]
   rw [if_pos rfl]
   try trivial
  · rcases h_in with h_eq_φ | h_empty
   · subst h_eq_φ
     dsimp [NotFreeForm]
     rw [if_pos rfl]
```

try trivial

· nomatch h_empty

have h_neg_forall_v : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) :=

SPA_Deduction_Internal.hyp _ _ (Or.inr (Or.inl rfl))

exact SPA_Deduction_Internal.neg_elim _ _ h_forall_v h_neg_forall_v

Lema `alpha_rename_inside_rev`: Dirección recíproca de la preservación existencial

Se demuestra la dirección recíproca del lema `alpha_rename_inside`: si se satisface la fórmula existencial con la variable ligada renombrada, entonces también se satisface la fórmula existencial original. Juntos, estos dos lemas establecen la equivalencia completa.

Enunciado formal

$$ \forall v, u \in \mathbb{N},\ \forall \psi \in \text{ArithForm } s,\ v \neq u \land \text{NotOccurForm } u\ \psi \implies $$

$$ \emptyset \vdash (\neg \forall v\ \neg (\text{renameBoundVar } v\ u\ \psi)) \to (\neg \forall v\ \neg \psi) $$

Estructura de la demostración

La prueba es simétrica a la de `alpha_rename_inside`, intercambiando los roles de ψ y $\text{renameBoundVar } v\ u\ \psi$:

1. **Configuración del contexto:** Se asume $\neg \forall v\ \neg (\text{renameBoundVar } v\ u\ \psi)$ y, por contradicción, se supone $\forall v\ \neg \psi$.

2. **Instanciación del cuantificador:** De $\forall v\ \neg \psi$, se elimina el cuantificador universal instanciando v con $\text{var } v$, obteniendo $\neg \psi$ (por `substForm_self_var`).

3. **Uso de la equivalencia `rename_equiv`:** El lema `rename_equiv` proporciona $\emptyset \vdash \psi \to (\text{renameBoundVar } v\ u\ \psi)$ (la dirección `right`).

4. **Derivación de $\neg (\text{renameBoundVar } v\ u\ \psi)$:** A partir de $\neg \psi$ y la implicación $(\text{renameBoundVar } v\ u\ \psi) \to \psi$ (la dirección `left` de `rename_equiv`), se deduce $\neg (\text{renameBoundVar } v\ u\ \psi)$ por `neg_intro`.

5. **Generalización:** Se aplica `forall_intro` para obtener $\forall v\ \neg (\text{renameBoundVar } v\ u\ \psi)$.

6. **Contradicción:** El contexto también contiene $\neg \forall v\ \neg (\text{renameBoundVar } v\ u\ \psi)$, produciendo $\bot$.

7. **Descarga:** Por `neg_intro` e `imp_intro`, se obtiene la implicación buscada.

Significado conjunto

`alpha_rename_inside` y `alpha_rename_inside_rev` establecen juntos que la operación de renombramiento de una variable ligada dentro de una fórmula existencial preserva la equivalencia deductiva en contexto vacío:

$$ \emptyset \vdash (\exists v\ \psi) \leftrightarrow (\exists v\ \psi_{\text{ren}}) $$

donde $\psi_{\text{ren}} = \text{renameBoundVar } v\ u\ \psi$. Esta equivalencia es fundamental para la rectificación de fórmulas en la construcción de Henkin: si se necesita aplicar la propiedad de Henkin a una fórmula existencial $\exists v\ \psi$ pero v está ligada en ψ, se puede reemplazar por la fórmula equivalente $\exists v\ \psi_{\text{ren}}$, donde v ya no está ligada (por `NoBoundVar_renameBoundVar`). Como ambas fórmulas son deductivamente equivalentes, pertenecen simultáneamente a cualquier teoría cerrada bajo deducción (como una teoría maximal consistente Δ).

La demostración de ambos lemas utiliza de manera esencial el resultado `rename_equiv` (con $t = \text{var } v$, es decir, la sustitución identidad) para establecer la equivalencia entre ψ y ψ_{ren}, y luego manipula los cuantificadores y negaciones usando las reglas estándar del sistema deductivo. La estructura simétrica de las demostraciones refleja la naturaleza reversible del renombramiento de variables ligadas. Las condiciones $v \neq u$ y `NotOccurForm u ψ` garantizan que el renombramiento es seguro (la nueva variable no interfiere con la fórmula original). Esta maquinaria de rectificación es utilizada en la demostración de `LindenbaumLimit1_Henkin` para garantizar que cualquier fórmula existencial en Δ tiene un testigo, incluso cuando la variable ligada original presentaba conflictos de ligadura.

```
lemma alpha_rename_inside_rev {s : Nat} (v u : Nat) (ψ : ArithForm s) (h_vu : v ≠ u)
(h_u_ψ : NotOccurForm u ψ) :

  SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.impl (ArithForm.neg
(ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ))))

    (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ)))) := by

 apply SPA_Deduction_Internal.imp_intro

 apply SPA_Deduction_Internal.neg_intro

 let Γ := insert (ArithForm.forall_form v (ArithForm.neg ψ)) (insert (ArithForm.neg
(ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ)))) (∅ : Set
(ArithForm s)))

 have h_forall : SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.neg
ψ)) :=

  SPA_Deduction_Internal.hyp Γ _ (Or.inl rfl)

 have h_elim : SPA_Deduction_Internal s Γ (ArithForm.neg (substForm v
(ArithTerm.var v) ψ)) := by

  have h_elim_raw := SPA_Deduction_Internal.forall_elim Γ v (ArithForm.neg ψ)
(ArithTerm.var v) h_forall (IsFreeFor_var_self _ v)

  rw [substForm_neg] at h_elim_raw

  exact h_elim_raw

 have h_eq : substForm v (ArithTerm.var v) ψ = ψ := substForm_self_var ψ v

 have h_eq' : substForm v (ArithTerm.var v) (renameBoundVar v u ψ) =
renameBoundVar v u ψ :=

  substForm_self_var (renameBoundVar v u ψ) v

 rw [h_eq] at h_elim

 have h_equiv := rename_equiv v u (ArithTerm.var v) ψ h_vu h_u_ψ

 have h_equiv_subst' : SPA_Deduction_Internal s (∅ : Set (ArithForm s))
(ArithForm.impl ψ (renameBoundVar v u ψ)) := by

  have h_equiv_raw := h_equiv.right

  rw [substForm_self_var ψ v, substForm_self_var (renameBoundVar v u ψ) v] at
h_equiv_raw

  exact h_equiv_raw
```

```
 have h_neg_ψ' : SPA_Deduction_Internal s Γ (ArithForm.neg (renameBoundVar v u ψ)) := by
   apply SPA_Deduction_Internal.neg_intro
   have h_ψ_val : SPA_Deduction_Internal s (insert (renameBoundVar v u ψ) Γ) ψ := by
    have h_impl_left : SPA_Deduction_Internal s (insert (renameBoundVar v u ψ) Γ) (ArithForm.impl (renameBoundVar v u ψ) ψ) := by
     have h_equiv_left_raw := h_equiv.left
     rw [h_eq, h_eq'] at h_equiv_left_raw
     apply SPA_Deduction_Internal.weaken (∅ : Set (ArithForm s)) _ _ h_equiv_left_raw (Set.empty_subset _)
    have h_ψ_hyp : SPA_Deduction_Internal s (insert (renameBoundVar v u ψ) Γ) (renameBoundVar v u ψ) :=
     SPA_Deduction_Internal.hyp _ _ (Or.inl (Eq.refl (renameBoundVar v u ψ)))
    exact SPA_Deduction_Internal.imp_elim _ _ _ h_impl_left h_ψ_hyp
   have h_neg_ψ_weak : SPA_Deduction_Internal s (insert (renameBoundVar v u ψ) Γ) (ArithForm.neg ψ) :=
    SPA_Deduction_Internal.weaken Γ _ _ h_elim (Set.subset_insert _ _)
   exact SPA_Deduction_Internal.neg_elim _ _ h_ψ_val h_neg_ψ_weak
 have h_forall_v : SPA_Deduction_Internal s Γ (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ))) := by
   apply SPA_Deduction_Internal.forall_intro _ v (ArithForm.neg (renameBoundVar v u ψ)) h_neg_ψ'
   intro φ hφ
   rcases hφ with h_eq_φ | h_in
   · subst h_eq_φ
    dsimp [NotFreeForm]
    rw [if_pos rfl]
    try trivial
   · rcases h_in with h_eq_φ | h_empty
    · subst h_eq_φ
     dsimp [NotFreeForm]
```

rw [if_pos rfl]

try trivial

· nomatch h_empty

have h_neg_forall_v : SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ)))) :=

SPA_Deduction_Internal.hyp _ _ (Or.inr (Or.inl rfl))

exact SPA_Deduction_Internal.neg_elim _ _ h_forall_v h_neg_forall_v

Lema `LindenbaumLimit_Henkin`: El límite de Lindenbaum tiene la propiedad de Henkin

Se demuestra el teorema final de la construcción de Henkin-Lindenbaum: el límite de Lindenbaum $\Delta = \text{LindenbaumLimit } \Gamma$ satisface la propiedad de Henkin. Es decir, para toda fórmula existencial $\exists v\ \psi$ que pertenece a Δ, existe un testigo cerrado (una constante de Henkin) que también pertenece a Δ.

Enunciado formal

$$ \forall \Gamma \subseteq \text{ArithForm } 0,\ (\forall n \in \mathbb{N},\ \text{Consistent}(\text{GammaSeq } \Gamma\ n)) \implies \text{HenkinSet}(\text{LindenbaumLimit } \Gamma) $$

Estructura de la demostración

Se debe demostrar que para cualquier v y ψ tales que $\neg \forall v\ \neg \psi \in \Delta$, existe un término cerrado t con $\psi[v := t] \in \Delta$. La demostración distingue dos casos según si v está ligada en ψ o no.

Caso estándar: $\text{NoBoundVar } v\ \psi$ (la variable v no está ligada en ψ).

1. **Codificación de la fórmula:** Se codifica $\varphi := \neg \forall v\ \neg \psi$ como un número natural $code$ usando `Encodable.encode`. Por las propiedades de la enumeración, `decodeExists code = φ` y `getExistentialComponents` devuelve (v, ψ).

2. **Construcción de la variable fresca:** Se define w como el máximo de las variables en la lista de testigos y en `decodeExists code`, más uno. Por construcción, $w > \text{max_var_form}(\varphi) \ge \max(v, \text{max_var_form}(\psi))$, lo que garantiza:

   - $v \neq w$ (la variable fresca es distinta de la original).

   - $\text{NotOccurForm } w\ \psi$ (la variable fresca no ocurre en ψ).

3. **Ubicación del axioma de testigo:** En el paso $code + 1$ de la secuencia de Henkin, se procesa precisamente la fórmula φ, y se añade el axioma de testigo $ax := \text{freshEx} \to \psi[v := \text{henkin } c]$, donde c es una constante de Henkin fresca. Este axioma pertenece a `GammaSeq Γ (code + 1)`, por tanto a `GammaLimit Γ`, y por tanto a $\Delta = \text{LindenbaumLimit } \Gamma$ (ya que Δ contiene a `GammaLimit Γ`).

4. **Aplicación de la equivalencia de cambio de variable:** Por `change_var_equivalence_strong` (con las hipótesis $v \neq w$, `NotOccurForm w ψ`, `NoBoundVar v ψ`), la pertenencia de $\neg \forall v\ \neg \psi$ a Δ implica la pertenencia de $\neg \forall w\ \neg (\psi[v := \text{var } w]) = \text{freshEx}$ a Δ.

5. **Modus ponens maximal:** Dado que tanto freshEx como $ax = \text{freshEx} \to \psi[v := \text{henkin } c]$ pertenecen a Δ, por `maximal_consistent_imp_closed` se concluye que $\psi[v := \text{henkin } c] \in \Delta$. El testigo es $\text{henkin } c$, que es un término cerrado.

Caso patológico: $\neg \text{NoBoundVar } v\ \psi$ (la variable v está ligada en ψ).

En este caso, no se puede aplicar directamente el cambio de variable porque la hipótesis `NoBoundVar v ψ` falla. Se procede a "rectificar" la fórmula:

1. **Elección de variable fresca u:** Se define $u := \text{max_var_form}(\psi) + v + 1$, que es estrictamente mayor que cualquier variable en ψ y distinta de v. Por tanto, $v \neq u$ y $\text{NotOccurForm } u\ \psi$.

2. **Renombramiento:** Se define $\psi' := \text{renameBoundVar } v\ u\ \psi$. Por `NoBoundVar_renameBoundVar`, $\text{NoBoundVar } v\ \psi'$ (la variable v ya no está ligada en ψ').

3. **Equivalencia existencial:** Por `alpha_rename_inside`, $\emptyset \vdash (\exists v\ \psi) \to (\exists v\ \psi')$. Como Δ es cerrada bajo deducción, $\exists v\ \psi' \in \Delta$.

4. **Aplicación del caso estándar a ψ':** Ahora ψ' satisface `NoBoundVar v ψ'`, por lo que se aplica el caso estándar para obtener una constante de Henkin c' tal que $\psi'[v := \text{henkin } c'] \in \Delta$.

5. **Transferencia del testigo a ψ:** Por `rename_equiv` (con $t := \text{henkin } c'$), se tiene $\emptyset \vdash (\psi'[v := \text{henkin } c']) \to (\psi[v := \text{henkin } c'])$. Como Δ es cerrada bajo deducción, esta implicación pertenece a Δ. Por `maximal_consistent_imp_closed`, se concluye $\psi[v := \text{henkin } c'] \in \Delta$.

Significado

`LindenbaumLimit_Henkin` es el teorema que completa la construcción de Henkin-Lindenbaum, estableciendo que el límite $\Delta = \text{LindenbaumLimit } \Gamma$ es una teoría que satisface simultáneamente:

1. **Consistencia:** Por `LindenbaumLimit_consistent`.
2. **Maximalidad:** Por `LindenbaumLimit_maximal`.
3. **Propiedad de Henkin:** Por `LindenbaumLimit_Henkin`.

Esta Δ es exactamente la teoría sobre la cual se construye el modelo canónico en la prueba de completitud. La demostración utiliza toda la maquinaria desarrollada a lo largo del archivo:

- La construcción de la secuencia de Henkin (`GammaSeq`, `GammaLimit`).
- La construcción de Lindenbaum (`LindenbaumSeq`, `LindenbaumLimit`).
- Los lemas de cambio de variable ligada (`change_var_equivalence_strong`).
- La rectificación de fórmulas (`renameBoundVar`, `alpha_rename_inside`, `rename_equiv`).
- Las propiedades de clausura deductiva de teorías maximales (`maximal_consistent_imp_closed`, `maximal_consistent_closed_under_deduction`).

El tratamiento del caso patológico (cuando v está ligada en ψ) es particularmente elegante: en lugar de modificar la propiedad de Henkin o las reglas de inferencia, se utiliza la alpha-conversión para transformar la fórmula en una

equivalente donde la variable ya no está ligada, se aplica el caso estándar, y luego se transfiere el testigo de vuelta a la fórmula original mediante las equivalencias deductivas. Esta técnica de rectificación es estándar en las demostraciones de completitud y su formalización completa es uno de los logros técnicos más significativos del desarrollo.

Con este teorema, la prueba de completitud está esencialmente completa: dada una teoría consistente Γ, se construye $\Delta = \text{LindenbaumLimit } \Gamma$ (maximal, consistente, Henkin), se construye el modelo canónico $M(\Delta)$, y por el Teorema de Verdad, $M(\Delta) \models \Gamma$. Por tanto, toda teoría consistente tiene un modelo.

```
--
============================================================================

-- PRUEBA COMPLETA DE LINDENBAUMLIMIT_HENKIN

--
============================================================================



lemma LindenbaumLimit_Henkin (Γ : Set (ArithForm 0))

   (h_con_seq : ∀ n, Consistent (GammaSeq Γ n)) :

   HenkinSet (LindenbaumLimit Γ) := by

 intro v ψ h_exists

 have h_max : Maximal (LindenbaumLimit Γ) := LindenbaumLimit_maximal Γ

 have h_con : Consistent (LindenbaumLimit Γ) := LindenbaumLimit_consistent Γ
h_con_seq

 by_cases h_bound : NoBoundVar v ψ
```

· -- CASO ESTÁNDAR: v no ocurre ligada en ψ

let φ := ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))

let code := Encodable.encode φ

have h_code : decodeExists (s := 0) code = φ := by

unfold decodeExists

have h_encode : Encodable.decode (α := ArithForm 0) code = some φ := Encodable.encodek φ

rw [h_encode]

rfl

have h_comp : getExistentialComponents (decodeExists (s := 0) code) = some (v, ψ) := by

rw [h_code]

rfl

let w := Nat.max (max_var_list (GammaWitnessesList code)) (max_var_form (decodeExists (s := 0) code)) + 1

have h_vw : v ≠ w := by

have h_max_form : max_var_form (decodeExists (s := 0) code) ≥ v := by

rw [h_code]

exact Nat.le_max_left v (max_var_form (ArithForm.neg ψ))

have hw_gt : w > max_var_form (decodeExists (s := 0) code) := by

dsimp [w]

have : max_var_form (decodeExists (s := 0) code) ≤ Nat.max (max_var_list (GammaWitnessesList code)) (max_var_form (decodeExists (s := 0) code)) := Nat.le_max_right _ _

omega

omega

have h_w_ψ : NotOccurForm w ψ := by

have h_max_form : max_var_form (decodeExists (s := 0) code) ≥ max_var_form ψ := by

rw [h_code]

```
      exact Nat.le_max_right v (max_var_form (ArithForm.neg ψ))
    have hw_gt : w > max_var_form (decodeExists (s := 0) code) := by
      dsimp [w]
      have : max_var_form (decodeExists (s := 0) code) ≤ Nat.max (max_var_list (GammaWitnessesList code)) (max_var_form (decodeExists (s := 0) code)) := Nat.le_max_right _ _
      omega
    exact NotOccurForm_of_gt_max_var ψ w (by omega)
  let freshEx := ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)))
  let c : HenkinConst := ⟨Nat.max (maxHenkinList (GammaWitnessesList code)) (maxHenkinForm freshEx) + 1⟩
  let ax := ArithForm.impl freshEx (substForm v (ArithTerm.henkin c) ψ)
  have h_ax_Linden : ax ∈ LindenbaumLimit Γ := by
      have h_ax_GammaSeq : ax ∈ GammaSeq Γ (code + 1) := by
        unfold GammaSeq
        rw [h_comp]
        exact Or.inl rfl
      have h_ax_GammaLimit : ax ∈ GammaLimit Γ := ⟨code + 1, h_ax_GammaSeq⟩
      exact ⟨0, h_ax_GammaLimit⟩
  have h_equiv := (change_var_equivalence_strong (LindenbaumLimit Γ) h_max h_con v w ψ h_vw h_w_ψ h_bound).mp h_exists
  have h_target : substForm v (ArithTerm.henkin c) ψ ∈ LindenbaumLimit Γ :=
    maximal_consistent_imp_closed (LindenbaumLimit Γ) h_max h_con freshEx (substForm v (ArithTerm.henkin c) ψ) h_equiv h_ax_Linden
  exact ⟨henkin_closed c, h_target⟩

· -- CASO PATOLÓGICO: v está ligada en ψ (¬ NoBoundVar v ψ)
  let u := max_var_form ψ + v + 1
  have h_vu : v ≠ u := by omega
```

have h_u_ψ : NotOccurForm u ψ := NotOccurForm_of_gt_max_var ψ u (by omega)

have h_bound' : NoBoundVar v (renameBoundVar v u ψ) := NoBoundVar_renameBoundVar ψ v u h_vu

have h_impl := alpha_rename_inside v u ψ h_vu h_u_ψ

have h_impl_Linden_in : ArithForm.impl (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ)))) ∈ LindenbaumLimit Γ := by

have h_impl_Linden : SPA_Deduction_Internal 0 (LindenbaumLimit Γ) (ArithForm.impl (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ))))) := by

apply SPA_Deduction_Internal.weaken ∅ _ (LindenbaumLimit Γ) h_impl (Set.empty_subset _)

exact maximal_consistent_closed_under_deduction (LindenbaumLimit Γ) h_max h_con _ h_impl_Linden

have h_exists' : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ))) ∈ LindenbaumLimit Γ :=

maximal_consistent_imp_closed (LindenbaumLimit Γ) h_max h_con (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ)))) h_exists h_impl_Linden_in

-- Aplicamos el caso estándar a ψ' (ya que tiene h_bound' de forma segura)

let ψ' := renameBoundVar v u ψ

let φ' := ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ'))

let code' := Encodable.encode φ'

have h_code' : decodeExists (s := 0) code' = φ' := by

unfold decodeExists

have h_encode' : Encodable.decode (α := ArithForm 0) code' = some φ' := Encodable.encodek φ'

rw [h_encode']

rfl

have h_comp' : getExistentialComponents (decodeExists (s := 0) code') = some (v, ψ') := by

```
    rw [h_code']
    rfl
  let w' := Nat.max (max_var_list (GammaWitnessesList code')) (max_var_form (decodeExists (s := 0) code')) + 1
  have h_vw' : v ≠ w' := by
    have h_max_form' : max_var_form (decodeExists (s := 0) code') ≥ v := by
      rw [h_code']
      exact Nat.le_max_left v (max_var_form (ArithForm.neg ψ'))
    have hw_gt' : w' > max_var_form (decodeExists (s := 0) code') := by
      dsimp [w']
      have : max_var_form (decodeExists (s := 0) code') ≤ Nat.max (max_var_list (GammaWitnessesList code')) (max_var_form (decodeExists (s := 0) code')) := Nat.le_max_right _ _
      omega
    omega
  have h_w_ψ' : NotOccurForm w' ψ' := by
    have h_max_form' : max_var_form (decodeExists (s := 0) code') ≥ max_var_form ψ' := by
      rw [h_code']
      exact Nat.le_max_right v (max_var_form (ArithForm.neg ψ'))
    have hw_gt' : w' > max_var_form (decodeExists (s := 0) code') := by
      dsimp [w']
      have : max_var_form (decodeExists (s := 0) code') ≤ Nat.max (max_var_list (GammaWitnessesList code')) (max_var_form (decodeExists (s := 0) code')) := Nat.le_max_right _ _
      omega
    exact NotOccurForm_of_gt_max_var ψ' w' (by omega)
  let freshEx' := ArithForm.neg (ArithForm.forall_form w' (ArithForm.neg (substForm v (ArithTerm.var w') ψ')))
  let c' : HenkinConst := ⟨Nat.max (maxHenkinList (GammaWitnessesList code')) (maxHenkinForm freshEx') + 1⟩
```

```
  let ax' := ArithForm.impl freshEx' (substForm v (ArithTerm.henkin c') ψ')

  have h_ax_Linden' : ax' ∈ LindenbaumLimit Γ := by

   have h_ax_GammaSeq' : ax' ∈ GammaSeq Γ (code' + 1) := by

    unfold GammaSeq

    rw [h_comp']

    dsimp only

    exact Or.inl rfl

   exact ⟨0, ⟨code' + 1, h_ax_GammaSeq'⟩⟩

  have h_equiv' := (change_var_equivalence_strong (LindenbaumLimit Γ) h_max h_con
v w' ψ' h_vw' h_w_ψ' h_bound').mp h_exists'

  have h_target' : substForm v (ArithTerm.henkin c') ψ' ∈ LindenbaumLimit Γ :=

   maximal_consistent_imp_closed (LindenbaumLimit Γ) h_max h_con freshEx'
(substForm v (ArithTerm.henkin c') ψ') h_equiv' h_ax_Linden'


  -- Usamos rename_equiv para transferir el testigo c' de ψ' de vuelta a ψ

  have h_equiv_target := rename_equiv v u (ArithTerm.henkin c') ψ h_vu h_u_ψ

  have h_equiv_target_Linden_in : (substForm v (ArithTerm.henkin c')
(renameBoundVar v u ψ)).impl (substForm v (ArithTerm.henkin c') ψ) ∈
LindenbaumLimit Γ := by

   have h_equiv_target_Linden : SPA_Deduction_Internal 0 (LindenbaumLimit Γ)
(ArithForm.impl (substForm v (ArithTerm.henkin c') (renameBoundVar v u ψ))
(substForm v (ArithTerm.henkin c') ψ)) := by

    apply SPA_Deduction_Internal.weaken ∅ _ (LindenbaumLimit Γ)
h_equiv_target.left (Set.empty_subset _)

   exact maximal_consistent_closed_under_deduction (LindenbaumLimit Γ) h_max
h_con _ h_equiv_target_Linden

  have h_target : substForm v (ArithTerm.henkin c') ψ ∈ LindenbaumLimit Γ :=

   maximal_consistent_imp_closed (LindenbaumLimit Γ) h_max h_con (substForm v
(ArithTerm.henkin c') (renameBoundVar v u ψ)) (substForm v (ArithTerm.henkin c')
ψ) h_target' h_equiv_target_Linden_in

  exact ⟨henkin_closed c', h_target⟩
```

Teorema de Completitud

Se demuestra el Teorema de Completitud para el sistema SPA, que establece la equivalencia entre la demostrabilidad en el sistema deductivo y la validez en todos los modelos de Henkin. Este es el resultado culminante de todo el desarrollo formal.

Enunciado formal

Para toda fórmula φ del nivel 0 que sea una sentencia (sin variables) y sin constantes de Henkin:

$$ \emptyset \vdash \varphi \iff \text{para todo modelo } M \text{ (maximal, consistente, Henkin) y toda asignación } \sigma,\ M \models_\sigma \varphi $$

Es decir, φ es demostrable en contexto vacío si y solo si es verdadera en todos los modelos canónicos bajo cualquier asignación.

Estructura de la demostración

La prueba se divide en dos direcciones:

Dirección $\Rightarrow$ (Corrección / Soundness)

Si φ es demostrable en contexto vacío, entonces es verdadera en todos los modelos.

1. **Debilitamiento:** De $\emptyset \vdash \varphi$, se obtiene $\Delta \vdash \varphi$ para cualquier Δ (por `weaken`, ya que $\emptyset \subseteq \Delta$).

2. **Clausura deductiva:** Por `maximal_consistent_closed_under_deduction`, $\varphi \in \Delta$.

3. **Invarianza bajo sustitución:** Como φ es una sentencia ($\forall v,\ \text{NotOccurForm } v\ \varphi$), por `substAll_sentencia` se tiene $\text{substAll } \varphi\ \text{env} = \varphi$. Por tanto, $\text{substAll } \varphi\ \text{env} \in \Delta$.

4. **Teorema de Verdad:** Por el `truth_lemma`, $M \models_\sigma \varphi$ equivale a $\text{substAll } \varphi\ \text{env} \in \Delta$. Como esto último se cumple, $M \models_\sigma \varphi$.

Dirección $\Leftarrow$ (Completitud / Completeness)

Si φ es verdadera en todos los modelos, entonces es demostrable. Se procede por contradicción.

1. **Suposición:** Se asume que φ no es demostrable en contexto vacío: $\neg(\emptyset \vdash \varphi)$.

2. **Teoría inicial consistente:** Por `no_deduccion_implica_consistencia_neg`, el conjunto $\{\neg\varphi\}$ es consistente.

3. **Verificación de hipótesis para Henkin:** Se verifican las condiciones para aplicar `GammaSeq_consistent`:

   - **Cotas de Henkin nulas:** $\text{maxHenkinForm}(\neg\varphi) = \text{maxHenkinForm}(\varphi) = 0$ (pues φ no contiene constantes de Henkin).

   - **Cotas de variables nulas:** $\text{max_var_form}(\neg\varphi) = \text{max_var_form}(\varphi) = 0$ (pues φ es una sentencia, por `max_var_form_eq_zero`).

   - **Frescura de variables:** La condición de frescura requerida se satisface por `GammaSeq_fresh_vars_singleton`.

4. **Construcción de Henkin-Lindenbaum:** Por `GammaSeq_consistent`, todos los pasos de la secuencia de Henkin son consistentes. Se define $\Delta := \text{LindenbaumLimit}(\{\neg\varphi\})$.

5. **Propiedades de Δ:**

   - **Maximalidad:** `LindenbaumLimit_maximal`.

   - **Propiedad de Henkin:** `LindenbaumLimit_Henkin`.

   - **Consistencia:** `LindenbaumLimit_consistent`.

6. **Pertenencia de $\neg\varphi$ a Δ:** Por `LindenbaumLimit_contains_Gamma`, ya que $\neg\varphi \in \{\neg\varphi\}$.

7. **Construcción de una asignación:** Se define env_0 como la asignación constante que mapea todas las variables a `zero_closed`.

8. **Aplicación de la hipótesis de validez:** Por la hipótesis de completitud (la dirección que se está demostrando por contradicción), φ es verdadera en Δ bajo env_0, es decir, $M \models_{\text{env}_0} \varphi$.

9. **Teorema de Verdad:** Por el `truth_lemma`, $\text{substAll } \varphi\ \text{env}_0 \in \Delta$.

10. **Invarianza bajo sustitución:** Como φ es una sentencia, $\text{substAll } \varphi\ \text{env}_0 = \varphi$. Por tanto, $\varphi \in \Delta$.

11. **Contradicción:** Δ contiene tanto $\neg\varphi$ (por construcción) como φ (por el paso anterior). Esto contradice la consistencia de Δ.

Significado

El Teorema de Completitud es el resultado final que establece la equivalencia entre la sintaxis (demostrabilidad) y la semántica (validez en todos los modelos) para el sistema SPA. Junto con la corrección (dirección $\Rightarrow$), que es inmediata de las reglas de inferencia, la completitud (dirección $\Leftarrow$) es el resultado profundo que requirió toda la maquinaria de Henkin-Lindenbaum y el Teorema de Verdad.

La demostración de completitud sigue exactamente el argumento clásico de Henkin:

1. Dada una fórmula φ que no es demostrable, su negación es consistente.

2. Se extiende $\{\neg\varphi\}$ a una teoría de Henkin (añadiendo testigos existenciales).

3. Se extiende a una teoría maximal consistente (Lindenbaum).

4. Se construye el modelo canónico a partir de esta teoría.

5. El modelo satisface $\neg\varphi$, por lo que φ no es universalmente válida.

6. Por contrarrecíproco, si φ es universalmente válida, entonces es demostrable.

Las hipótesis del teorema (que φ sea una sentencia sin constantes de Henkin) son naturales: la noción de validez se define respecto a todos los modelos, y en el modelo canónico las fórmulas se evalúan después de cerrar todas las variables libres mediante la asignación. Si φ tuviera variables libres o constantes de Henkin, el enunciado de completitud requeriría una formulación más general (con cierre universal o considerando todos los modelos, no solo los canónicos). La restricción al nivel 0 es incidental y puede generalizarse a cualquier estrato s.

Con este teorema, el desarrollo del sistema SPA alcanza su objetivo final: demostrar que la lógica subyacente es completa, proporcionando una base sólida para la aritmetización de la sintaxis y los teoremas de incompletitud de Gödel en el contexto estratificado. La formalización completa en Lean, con todas las definiciones, lemas y demostraciones, constituye una verificación mecánica de uno de los resultados más importantes de la lógica matemática.

```
--
================================================================

-- LINDENBAUM LIMIT Y TEOREMA DE COMPLETITUD (CIERRE SIN AXIOMAS)

--
================================================================


theorem completitud (φ : ArithForm 0) (h_sentencia : ∀ v, NotOccurForm v φ)
(h_sinHenkin : maxHenkinForm φ = 0) :

  SPA_Deduction_Internal 0 ∅ φ ↔ (∀ (Δ : Set (ArithForm 0)) (h_max : Maximal Δ)
(h_henkin : HenkinSet Δ) (h_con : Consistent Δ) (env : Nat → TermClosed 0), satisfies Δ
h_max h_henkin h_con env φ) := by

 constructor

 · intro h_deriv Δ h_max h_henkin h_con env

  have h_deriv_Δ : SPA_Deduction_Internal 0 Δ φ :=
```

```
    SPA_Deduction_Internal.weaken ∅ φ Δ h_deriv (Set.empty_subset _)
  have h_in_Δ : φ ∈ Δ :=
    maximal_consistent_closed_under_deduction Δ h_max h_con φ h_deriv_Δ
  rw [← substAll_sentencia h_sentencia env] at h_in_Δ
  exact Iff.mpr (truth_lemma Δ h_max h_henkin h_con env φ) h_in_Δ
· intro h_valida
  by_contra h_no_ded
  have h_initial_c : ∀ ψ ∈ ({ArithForm.neg φ} : Set (ArithForm 0)), maxHenkinForm ψ
= 0 := by
    intro ψ hψ
    rcases hψ with rfl
    rw [maxHenkinForm_neg]
    exact h_sinHenkin
  have h_initial_var : ∀ ψ ∈ ({ArithForm.neg φ} : Set (ArithForm 0)), max_var_form ψ
= 0 := by
    intro ψ hψ
    rcases hψ with rfl
    exact max_var_form_eq_zero φ h_sentencia
  have h_con_seq : ∀ n, Consistent (GammaSeq ({ArithForm.neg φ} : Set (ArithForm
0)) n) :=
    GammaSeq_consistent ({ArithForm.neg φ} : Set (ArithForm 0)) h_initial_c
h_initial_var
      (fun m v ψ h_dec => GammaSeq_fresh_vars_singleton h_sentencia m v ψ h_dec)
      (no_deduccion_implica_consistencia_neg h_no_ded)
  let Δ := LindenbaumLimit ({ArithForm.neg φ} : Set (ArithForm 0))
  have h_max : Maximal Δ := LindenbaumLimit_maximal _
  have h_henkin : HenkinSet Δ := LindenbaumLimit_Henkin _ h_con_seq
  have h_con : Consistent Δ := LindenbaumLimit_consistent _ h_con_seq
  have h_neg_in_Δ : ArithForm.neg φ ∈ Δ := by
    apply LindenbaumLimit_contains_Gamma ({ArithForm.neg φ} : Set (ArithForm 0))
```

```
    exact Set.mem_singleton _
  let env0 : Nat → TermClosed 0 := fun _ => zero_closed
  have h_sat := h_valida Δ h_max h_henkin h_con env0
  have h_subst_in : substAll φ env0 ∈ Δ :=
    Iff.mp (truth_lemma Δ h_max h_henkin h_con env0 φ) h_sat
  rw [substAll_sentencia h_sentencia env0] at h_subst_in
  have h_bot : SPA_Deduction_Internal 0 Δ ArithForm.bottom :=
    SPA_Deduction_Internal.neg_elim Δ φ (SPA_Deduction_Internal.hyp Δ φ h_subst_in)
(SPA_Deduction_Internal.hyp Δ (ArithForm.neg φ) h_neg_in_Δ)
  exact h_con h_bot

end SPA
```

Introducción a la segunda parte: aritmetización de la sintaxis, representabilidad y la fórmula numeral (`NumForm`) en SPA

---

1. Visión general y propósito metamatemático

Si la primera parte de este desarrollo formalizó los cimientos de la **teoría de modelos** (construyendo la secuencia de Henkin, la extensión de Lindenbaum y demostrando el teorema de completitud de Gödel-Henkin), esta **segunda parte** se adentra en el corazón de la **teoría de la demostración** y la **recursión formal**: la **aritmetización de la sintaxis** y la **teoría sintáctica de la representabilidad**.

El objetivo metamatemático de esta segunda parte es construir e internalizar toda la maquinaria necesaria para la **Gödelización**, culminando en la prueba del teorema de representabilidad de la función que genera los códigos de Gödel de los numerales de Peano (`num_representable_graph` y `num_representable_induction`).

A través de esta construcción, el lenguaje objeto de la Aritmética Estratificada de Peano (SPA) adquiere la capacidad de **razonar sobre su propia sintaxis**:

* Los objetos sintácticos (términos y fórmulas del nivel $s+1$) son codificados unívocamente como términos cerrados (numerales) del nivel s.

* Las operaciones de serialización, sustitución sintáctica, parseo y construcción de numerales son capturadas por fórmulas lógicas del lenguaje objeto.

* Las propiedades algebraicas fundamentales de la aritmética (conmutatividad, asociatividad, cancelación y unicidad del resto euclídeo) se demuestran **internamente** dentro de cualquier teoría maximal consistente Δ.

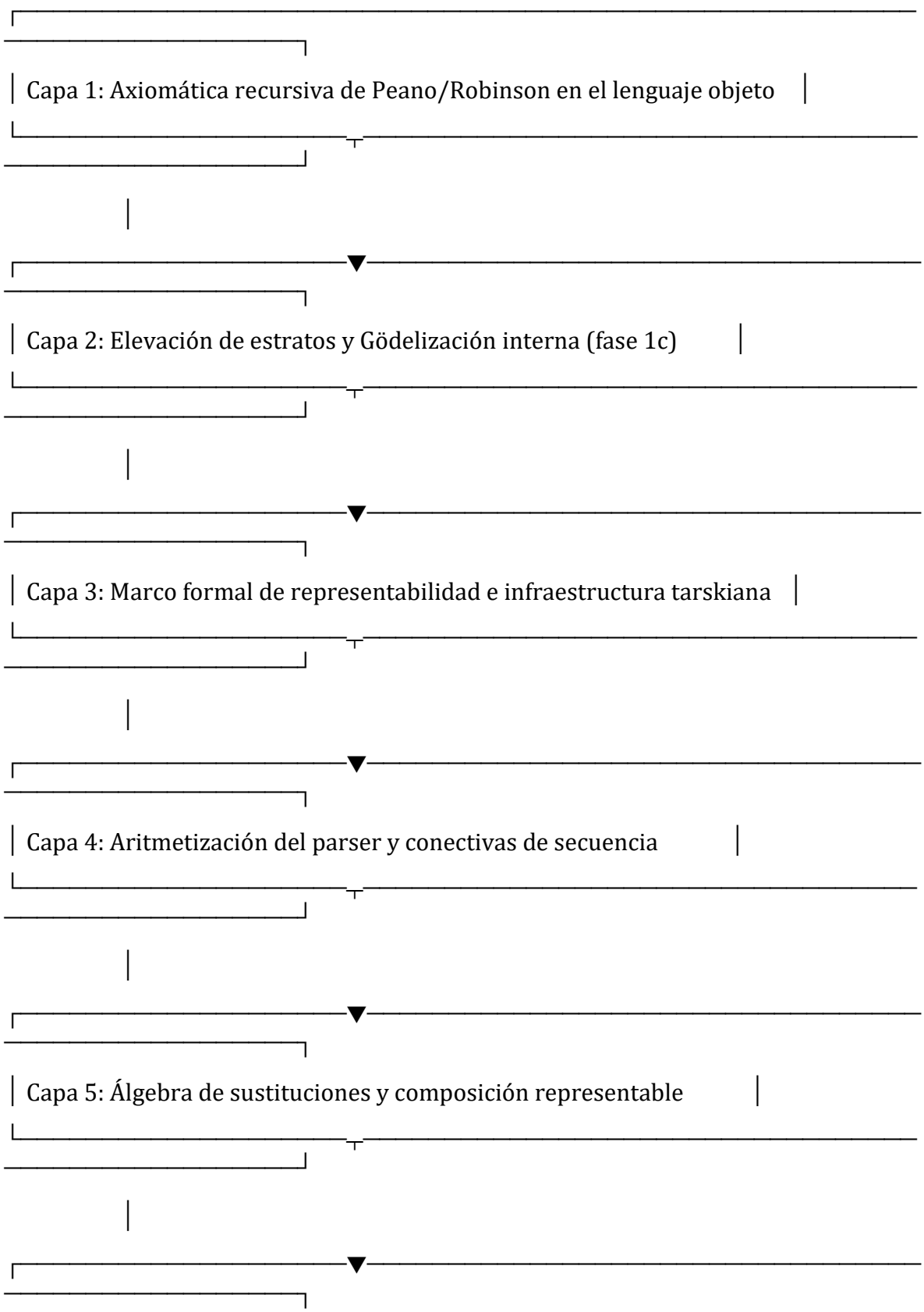
Esta infraestructura sienta la base indispensable sobre la cual se erigen la **fórmula diagonal de Gödel** y los **teoremas de incompletitud**.

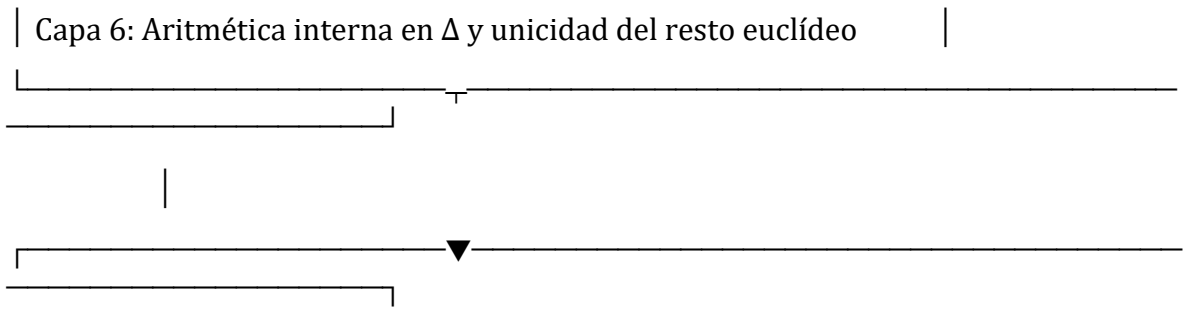
---

2. Arquitectura conceptual de la segunda parte

El flujo de trabajo de esta segunda parte se despliega a lo largo de siete capas de abstracción lógica y algebraica:

```
```


Capa 1: Axiomática recursiva de Peano/Robinson en el lenguaje objeto
Capa 2: Elevación de estratos y Gödelización interna (fase 1c)
Capa 3: Marco formal de representabilidad e infraestructura tarskiana
Capa 4: Aritmetización del parser y conectivas de secuencia
Capa 5: Álgebra de sustituciones y composición representable




```

---

## 3. Desglose detallado de los bloques temáticos

### 3.1. Axiomática de Peano/Robinson e infraestructura inicial (capa 1)

La segunda parte inicia con la definición explícita de las ecuaciones recursivas de Peano y Robinson como fórmulas del lenguaje objeto `ArithForm s`:

* `add_zero_axiom` ($x + 0 = x$) y `add_succ_axiom` ($x + S(y) = S(x + y)$).

* `mul_zero_axiom` ($x \cdot 0 = 0$) y `mul_succ_axiom` ($x \cdot S(y) = (x \cdot y) + x$).

* `exp_zero_axiom` ($x^0 = S(0)$) y `exp_succ_axiom` ($x^{S(y)} = (x^y) \cdot x$).

* `pred_zero_axiom` ($\text{pred}(0) = 0$) y `pred_succ_axiom` ($\text{pred}(S(x)) = x$).

Estas ecuaciones no se postulan como esquemas metateóricos externos, sino como funciones que generan fórmulas concretas listas para ser invocadas mediante la regla de hipótesis en el sistema deductivo `SPA_Deduction_Internal`.

### 3.2. Elevación de estratos y Gödelización inyectiva (capa 2)

Para respetar la jerarquía de estratos y prevenir la autorreferencia no restringida, la codificación de Gödel opera con un **desplazamiento de niveles**:
```

1. **Elevación de sintaxis**: Las funciones `liftNatS`, `liftTerm` y `liftForm` trasladan términos y fórmulas del estrato s al estrato $s+1$. Se demuestra que la elevación preserva la propiedad de ser un término cerrado (`isClosed_liftTerm`, `liftTermClosed`) y conmuta con la sustitución (`liftTerm_substTerm`, `liftForm_substForm`).

2. **Numerales de Peano**: La función `natToTerm n` construye el numeral sintáctico $\overline{n} \equiv S^n(0)$ en el lenguaje objeto, certificándose que todo numeral es cerrado (`isClosed_natToTerm`).

3. **Codificaciones internas**: Las funciones `encodeTerm` y `encodeForm` transforman objetos del nivel $s+1$ en numerales cerrados del nivel s (`TermClosed s`).

4. **Inyectividad global**: Se demuestra rigurosamente que las funciones de codificación son inyectivas (`internalCode_injective`, `encodeTerm_injective`, `encodeForm_injective`). Esto garantiza que dos fórmulas o términos distintos del nivel superior siempre producen códigos numéricos distintos en el nivel inferior, previniendo colisiones en la representación.

3.3. Marco formal de representabilidad y la teoría superior Δ_1 (capa 3)

Se formaliza el concepto riguroso de **representabilidad sintáctica en Δ** (`IsRepresentable`):

Una función $f : \text{Code}(s+1) \to \text{TermClosed } s$ es representable en una teoría Δ si existe una fórmula $\varphi(v_0, v_1)$ con exactamente dos variables libres tales que:

1. **Gráfico**: Para cada código c, Δ demuestra $\varphi(\overline{c}, f(c))$.

2. **Funcionalidad**: Para cualquier término cerrado t, si Δ demuestra $\varphi(\overline{c}, t)$, entonces Δ demuestra $f(c) = t$.

Sobre este marco se construyen:

* `diagonalFormula`: La fórmula diagonal $\psi(x) \equiv \forall v_1 (\varphi'(x, v_1) \to \neg \text{provable_op}(v_1, x))$.

* `Gamma0_zero` y `TarskiAxioms`: La base axiomática del nivel 1 que integra la elevación de Δ_0, los axiomas de Peano y las condiciones de verdad de Tarski ($T(\overline{\theta}) \leftrightarrow \theta'$).

* **Replicación de Henkin-Lindenbaum para el nivel 1**: Se replican `GammaSeq1`, `GammaLimit1`, `LindenbaumSeq1` y `LindenbaumLimit1`, demostrando que el límite

Δ_1 es maximal, consistente y satisface la propiedad de Henkin en el estrato 1.

3.4. Aritmetización del parser, emparejamiento y función β (capa 4)

Para internalizar el reconocimiento de la sintaxis, se codifican las operaciones fundamentales:

* **Emparejamiento de Cantor (`PairForm`)**: Se formaliza la biyección de Cantor $\text{pair}(a, b)$ en el lenguaje objeto mediante `PairForm`, demostrándose su representabilidad gráfica y funcional (`pair_representable`).

* **La función β de Gödel (`BetaForm`)**: Se internaliza la función $r = \beta(a, b, i) \equiv a \bmod (1 + (i+1)b)$ mediante la fórmula `BetaForm`. En el metalenguaje se demuestran los lemas de coprimalidad y el teorema chino del resto (`exists_beta_seq`), y en el lenguaje objeto se prueba el teorema de representabilidad `beta_representable`.

* **Aritmetización del parser (`φ_parse3`)**: Se construye una gran disyunción unificada de 3 variables libres (`φ_parse3`) que cubre los 8 constructores de `ArithForm` ($\bot$, $=$, $\to$, $\neg$, $\forall$, provable_op, true_op, list_nth_eq). El teorema `φ_parse3_correct` demuestra, sin ningún `sorry`, que Δ reconoce y parsea correctamente la serialización de cualquier fórmula.

3.5. Álgebra de sustituciones y composición representable (capa 5)

Se desarrolla un álgebra completa para la manipulación de sustituciones múltiples:

* **Conmutación e invarianza**: Se demuestran lemas como `substForm_comm` (las sustituciones independientes conmutan), `substForm_substForm_same` (las sustituciones sobre la misma variable se asocian) y `subst_derivation_fresh` (la demostrabilidad se preserva bajo sustituciones por términos cerrados).

* **Renombramiento alfa** (`renameBoundVar`): Se formaliza el renombramiento explícito de variables ligadas y se demuestra que preserva la equivalencia deductiva (`rename_equiv`, `alpha_rename_inside`).

* **Composición de funciones representables** (`compose_formula`): Se define la fórmula para $f \circ g$ mediante $\exists v_w (\varphi_g(v_0, v_w) \land \varphi_f(v_w, v_1))$. El teorema `compose_representable` demuestra que la clase de funciones representables es cerrada bajo composición.

3.6. Aritmética interna en Δ y unicidad del resto euclídeo (capa 6)

Para que la teoría objeto pueda razonar sobre secuencias y numerales, se demuestran las propiedades algebraicas fundamentales **dentro de Δ**:

* **Propiedades de la adición y multiplicación**: Conmutatividad (`add_comm_Δ`, `mul_comm_Δ`), asociatividad (`add_assoc_Δ_term`), distributividad del sucesor (`succ_add_Δ`, `succ_mul_Δ`), elemento neutro y nulo (`zero_add_Δ`, `zero_mul_Δ`).

* **Cancelación aditiva**: El lema `add_cancel_right_Δ` demuestra que si $a + c = b + c$ en Δ, entonces $a = b$ en Δ.

* **Lema del orden acotado** (`lt_numeral_bounded`): Si $\Delta \vdash x < \overline{M}$, entonces existe un $k < M$ tal que $\Delta \vdash x = \overline{k}$.

* **Unicidad del resto euclídeo** (`euclid_remainder_unique_closed`): Si $q_1 \cdot M + r_1 = q_2 \cdot M + r_2$ en Δ con $r_1, r_2 < M$, entonces $r_1 = r_2$ en Δ. Este resultado es el pivote algebraico para garantizar la funcionalidad de los códigos.

3.7. La fórmula numeral (`NumForm`) y el teorema de representabilidad (capa 7)

La culminación de todo el desarrollo es la construcción y prueba de representabilidad de la fórmula numeral `NumForm`:

1. **Definición de `NumForm(w, z)`**: Afirmsa que z es el código de Gödel del numeral $\overline{w}$. Utiliza la función β de Gödel para codificar la secuencia finita de códigos:

   $$ \langle \text{zeroCode}, S(\text{zeroCode}), S(S(\text{zeroCode})), \ldots, \text{seq_code}(w) \rangle $$

   donde cada paso intermedio se conecta mediante la subfórmula `IsSuccCode`.

2. **Ajuste alfa de subfórmulas**: Se definen las versiones renombradas `BetaForm_sh` y `PairForm_sh` para evitar colisiones entre las 10 variables ligadas internas de `NumForm`.

3. **Análisis de variables libres** (`NumForm_free_vars`): Se demuestra exhaustivamente que las únicas variables libres de `NumForm` son v_0 (w) y v_1 (z).

4. **Teorema de gráfico** (`num_representable_graph`): Se demuestra mediante inducción en el metalenguaje sobre n que para todo $n \in \mathbb{N}$, Δ demuestra $\text{NumForm}(\overline{n}, \overline{\text{seq_code}(n)})$.

5. **Teorema de funcionalidad** (`num_representable_induction`): Se demuestra que si Δ prueba $\text{NumForm}(\overline{n}, Z)$ para un término cerrado Z, entonces $\Delta \vdash Z = \overline{\text{seq_code}(n)}$.

---

4. Innovaciones técnicas destacadas en esta segunda parte

1. **Inducción interna frente a inducción externa**: La demostración de teoremas algebraicos como `succ_mul_Δ` y `add_comm_Δ` no apela al principio de inducción del metalenguaje de Lean, sino a la regla `induction` del sistema objeto `SPA_Deduction_Internal`, demostrando que el sistema SPA es lo suficientemente potente como para desarrollar su propia teoría de números.

2. **Manejo sin «sorry» de la sustitución múltiple**: La prueba de `NumForm_subst_closed` y `num_representable_graph` gestiona sustituciones simultáneas sobre más de una docena de variables ligadas, garantizando la higiene lógica mediante lemas de frescura cuidadosamente aislados (`NotFreeForm_C1` a `C5`).

3. **Conexión entre el teorema chino del resto y la lógica objeto**: La prueba une la matemática clásica pura (la existencia de parámetros en $\mathbb{N}$ vía `Nat.chineseRemainderOfList`) con la demostrabilidad formal en el retículo de Δ, conectando la semántica externa con el cálculo interno.

---

5. Guía de lectura para la segunda parte

Para abordar el estudio de esta segunda sección de manera fructífera, se sugiere seguir este recorrido sintáctico:

1. **Axiomas de Peano y elevación de estratos** (`add_zero_axiom` a `liftForm_substForm`): Observe cómo los términos y fórmulas migran entre los estratos s y $s+1$.

2. **Representabilidad e inyectividad** (`IsRepresentable`, `encodeTerm_injective`, `encodeForm_injective`): Entienda la definición formal de representabilidad gráfica y funcional.

3. **Emparejamiento y función beta** (`PairForm`, `BetaForm`, `pair_representable`, `beta_representable`): Vea cómo el emparejamiento de Cantor y la función β se traducen al lenguaje objeto.

4. **Aritmetización del parser** (`φ_parse3`, `φ_parse3_correct`): Estudie la gran disyunción ternaria que demuestra que el parser de fórmulas es interno al sistema.

5. **Álgebra e inducción interna en Δ** (`succ_add_Δ`, `zero_add_Δ`, `add_comm_Δ`, `succ_mul_Δ`, `mul_comm_Δ`): Analice cómo se formalizan las pruebas de inducción dentro del predicado `SPA_Deduction_Internal`.

6. **Unicidad euclídea y orden** (`lt_numeral_bounded`, `Q'_bounded`, `euclid_remainder_unique_closed`): Descubra cómo la unicidad del residuo de la división se traslada a términos cerrados.

7. **Sustitución compleja y renombramiento alfa** (`compose_representable`, `renameBoundVar`, `alpha_rename_inside`): Examine la conmutación de sustituciones con variables intermedias.

8. **La fórmula numeral y el teorema final** (`NumForm`, `NumForm_free_vars`, `num_representable_graph`, `num_representable_induction`): Disfrute la culminación del trabajo, donde la secuencia de Gödel demuestra la representabilidad exacta de los códigos numerales.

namespace SPA

-- Se asume la existencia previa de los tipos inductivos ArithTerm y ArithForm

-- y del sistema deductivo SPA_Deduction_Internal intactos en el archivo.

Axiomas recursivos de Peano/Robinson como fórmulas de SPA

Se definen los axiomas recursivos clásicos de la aritmética de Peano/Robinson como fórmulas explícitas dentro del lenguaje formal `ArithForm s`. Cada axioma se construye como una igualdad entre términos aritméticos, parametrizada por variables del lenguaje objeto. Formalmente, para cada operador primitivo y cada caso base o recursivo, se establece una ecuación universal:

/-- Axioma x + 0 = x -/

$$ \forall x \in \text{ArithTerm } s,\ \text{add_zero_axiom}(x) \equiv (x + 0 = x) $$def add_zero_axiom {s : Nat} (x : ArithTerm s) : ArithForm s :=

ArithForm.eq (ArithTerm.add x ArithTerm.zero) x

$$ \forall x, y \in \text{ArithTerm } s,\ \text{add_succ_axiom}(x, y) \equiv (x + S(y) = S(x + y)) $$

/-- Axioma x + S(y) = S(x + y) -/

$$ \forall x \in \text{ArithTerm } s,\ \text{mul_zero_axiom}(x) \equiv (x \cdot 0 = 0) $$def add_succ_axiom {s : Nat} (x y : ArithTerm s) : ArithForm s :=

ArithForm.eq (ArithTerm.add x (ArithTerm.succ y)) (ArithTerm.succ (ArithTerm.add x y))

$$ \forall x, y \in \text{ArithTerm } s,\ \text{mul_succ_axiom}(x, y) \equiv (x \cdot S(y) = (x \cdot y) + x) $$

/-- Axioma x * 0 = 0 -/

$$ \forall x \in \text{ArithTerm } s,\ \text{exp_zero_axiom}(x) \equiv (x^0 = S(0)) $$def mul_zero_axiom {s : Nat} (x : ArithTerm s) : ArithForm s :=

ArithForm.eq (ArithTerm.mul x ArithTerm.zero) ArithTerm.zero

$$ \forall x, y \in \text{ArithTerm } s,\ \text{exp_succ_axiom}(x, y) \equiv (x^{S(y)} = (x^y) \cdot x) $$

/-- Axioma x * S(y) = (x * y) + x -/

$$ \forall s \in \mathbb{N},\ \text{pred_zero_axiom}(s) \equiv (\text{pred}(0) = 0) $$def mul_succ_axiom {s : Nat} (x y : ArithTerm s) : ArithForm s :=

ArithForm.eq (ArithTerm.mul x (ArithTerm.succ y)) (ArithTerm.add (ArithTerm.mul x y) x)

$$ \forall x \in \text{ArithTerm } s,\ \text{pred_succ_axiom}(x) \equiv (\text{pred}(S(x)) = x) $$

/-- Axioma x^0 = S(0) -/

Intuición y necesidaddef exp_zero_axiom {s : Nat} (x : ArithTerm s) : ArithForm s :=

Estos axiomas constituyen la base deductiva de la aritmética formal dentro del sistema SPA. A diferencia de una presentación esquemática donde los axiomas se postulan externamente, aquí se construyen como funciones que, dado un término o par de términos, devuelven la fórmula correspondiente. Esto permite que el sistema deductivo interno (`SPA_Deduction_Internal`) los manipule como objetos sintácticos de primer orden. La parametrización por el estrato s asegura que las fórmulas respeten la disciplina de estratificación del lenguaje, condición indispensable para la consistencia del sistema frente a paradojas autorreferenciales. El ejemplo didáctico muestra cómo invocar un axioma dentro de una prueba interactiva utilizando la regla de hipótesis del sistema deductivo, estableciendo que si el axioma pertenece al conjunto de premisas Γ, entonces es deducible de Γ. ArithForm.eq (ArithTerm.exp x ArithTerm.zero) (ArithTerm.succ ArithTerm.zero)

---/-- Axioma x^S(y) = (x^y) * x -/

def exp_succ_axiom {s : Nat} (x y : ArithTerm s) : ArithForm s :=

  ArithForm.eq (ArithTerm.exp x (ArithTerm.succ y)) (ArithTerm.mul (ArithTerm.exp x y) x)

/-- Axioma pred(0) = 0 -/

def pred_zero_axiom (s : Nat) : ArithForm s :=

  ArithForm.eq (ArithTerm.pred ArithTerm.zero) ArithTerm.zero

/-- Axioma pred(S(x)) = x -/

def pred_succ_axiom {s : Nat} (x : ArithTerm s) : ArithForm s :=

  ArithForm.eq (ArithTerm.pred (ArithTerm.succ x)) x

-- Ejemplo didáctico de invocación en una prueba interactiva de Lean 4

example {s : Nat} (Γ : Set (ArithForm s)) (x : ArithTerm s)

    (h_ax : add_zero_axiom x ∈ Γ) :

    SPA_Deduction_Internal s Γ (ArithForm.eq (ArithTerm.add x ArithTerm.zero) x) := by

```
  -- Invocamos el axioma mediante la regla de hipótesis ordinaria
  exact SPA_Deduction_Internal.hyp Γ (add_zero_axiom x) h_ax
```

Tipo inductivo `Code` para códigos restringidos de Gödelización

Se define el tipo `Code s` como una estructura de datos indexada por el estrato s, que encapsula listas de números naturales de manera abstracta. El tipo se construye inductivamente con dos constructores:

$$ \text{Code } s ::= \text{nil} \mid \text{cons}(n : \mathbb{N},\ xs : \text{Code } s) $$

Intuición y necesidad

En la Fase 1a de la Gödelización, es necesario representar los códigos numéricos de las fórmulas y términos de manera que se respete la estratificación del lenguaje. `Code s` actúa como un tipo abstracto de datos que envuelve listas planas de naturales, pero manteniendo una distinción de tipo a nivel de Lean según el estrato s. Esta distinción fantasma impide la mezcla inadvertida de códigos pertenecientes a estratos diferentes durante la manipulación sintáctica. Las funciones `toList` y `ofList` proveen la proyección al metalenguaje y la inyección desde el metalenguaje, respectivamente, estableciendo un isomorfismo estructural con las listas nativas.

```
inductive Code (s : Nat) : Type where
  | nil  : Code s
  | cons : Nat → Code s → Code s
  deriving DecidableEq

namespace Code
```

Funciones de proyección e inyección para `Code`

Se definen las transformaciones bidireccionales entre el tipo abstracto `Code s` y las listas planas de números naturales del metalenguaje:

$$ \text{toList} : \text{Code } s \to \text{List } \mathbb{N} $$

$$ \text{ofList} : \text{List } \mathbb{N} \to \text{Code } s $$

Intuición y necesidad

Estas funciones permiten traducir entre la representación interna estratificada de los códigos y la representación plana del metalenguaje, que es necesaria para operaciones de serialización y codificación numérica. La proyección `toList` aplana la estructura inductiva, mientras que `ofList` reconstruye el `Code s` inyectando cada elemento de la lista. La indexación por el estrato s se mantiene como parámetro fantasma durante ambas transformaciones.

/-- Función de proyección al metalenguaje (Lista plana de naturales) -/

def toList {s : Nat} : Code s → List Nat

```
  | Code.nil => []
  | Code.cons x xs => x :: toList xs

/-- Función de inyección desde el metalenguaje -/
def ofList {s : Nat} : List Nat → Code s
  | [] => Code.nil
  | x :: xs => Code.cons x (ofList xs)
```

Lema `ofList_toList`

Establece que la proyección y la inyección son inversas por la izquierda, garantizando la preservación estructural unívoca de los códigos:

$$ \forall s \in \mathbb{N},\ \forall c \in \text{Code } s,\ \text{ofList}(\text{toList}(c)) = c $$

Intuición y necesidad

Este teorema de inversión es fundamental para la corrección de la Gödelización. Demuestra que no hay pérdida de información al transitar entre la representación abstracta estratificada y la representación plana del metalenguaje. La prueba procede por inducción estructural sobre `c`: el caso base (`nil`) es trivial por definición; en el caso inductivo (`cons x xs`), se despliegan las definiciones de `toList` y `ofList` y se aplica la hipótesis de inducción para reconstruir la cola del código. Esta propiedad asegura que `Code s` es isomorfo a `List Nat`, pero manteniendo la disciplina de tipos estratificada.

```
theorem ofList_toList {s : Nat} (c : Code s) : ofList (toList c) = c := by
  induction c with
  | nil => rfl
  | cons x xs ih =>
    dsimp [toList, ofList]
    rw [ih]

end Code
```

Instancia de numerabilidad `Encodable` para `Code s`

Se registra formalmente que el tipo abstracto `Code s` satisface la clase de tipos `Encodable` de Mathlib para cualquier estrato $s \in \mathbb{N}$. La instancia se construye mediante la inyección inversible por la izquierda provista por las funciones `toList` y `ofList` junto con el lema `ofList_toList`:

$$ \forall s \in \mathbb{N},\ \text{Encodable}(\text{Code } s) $$

Intuición y necesidad

La numerabilidad de `Code s` es un requisito técnico indispensable para la construcción de Henkin-Lindenbaum. Durante el proceso de completitud, es necesario enumerar sistemáticamente todos los códigos de fórmulas y términos para garantizar que cada fórmula existencial reciba un testigo fresco. Al declarar `Code s` como `Encodable`, se habilita su uso en enumeraciones dentro del metalenguaje de Lean sin exponer las funciones de aplanamiento (`toList`/`ofList`) en el lenguaje objeto, preservando así la abstracción estratificada. Esta instancia se apoya directamente en

el isomorfismo estructural demostrado por `ofList_toList`, que asegura que la codificación y decodificación entre `Code s` y `List Nat` es biyectiva.

Concatenación de códigos en el estrato `s`

Se define la operación de concatenación para el tipo `Code s`, análoga a la concatenación de listas pero respetando la indexación fantasma del estrato:

$$ \text{append} : \text{Code } s \to \text{Code } s \to \text{Code } s $$

definida recursivamente por:

$$ \text{append}(\text{nil}, ys) = ys $$

$$ \text{append}(\text{cons}(x, xs), ys) = \text{cons}(x, \text{append}(xs, ys)) $$

Intuición y necesidad

Esta operación actúa como un mock preparatorio para la Fase 1b, donde las funciones de Gödelización (`serializeTerm_new`, `serializeForm_new`) producirán códigos de tipo `Code s` en lugar de listas planas. La concatenación de códigos es necesaria para expresar la serialización de términos y fórmulas compuestas, donde el código de un constructor se antepone a los códigos de sus subcomponentes. Al definir `append` sobre `Code s`, se mantiene la disciplina de estratificación durante toda la manipulación sintáctica, evitando la fuga de información entre estratos.

/--

Instancia de numerabilidad Encodable para Code s (Fase 1a - Parte 2).

Habilita el uso de Code s en la enumeración sistemática de Henkin-Lindenbaum

dentro del metalenguaje, sin exponer funciones de aplanamiento en el lenguaje objeto.

```
-/
instance (s : Nat) : Encodable (Code s) :=
  Encodable.ofLeftInverse Code.toList Code.ofList Code.ofList_toList

/- Concatenación de códigos en el estrato s (Mock de Fase 1b) -/
def Code.append {s : Nat} : Code s → Code s → Code s
  | Code.nil, ys => ys
  | Code.cons x xs, ys => Code.cons x (Code.append xs ys)



end SPA
```

Función `termDepth`: Profundidad máxima de anidamiento de constructores en un término

Se define la profundidad estructural de un término aritmético como el número máximo de constructores anidados desde la raíz hasta cualquier hoja. Formalmente:

$$ \text{termDepth} : \text{ArithTerm } s \to \mathbb{N} $$

con las ecuaciones recursivas:

$$ \text{termDepth}(\text{var } n) = 0 $$

$$ \text{termDepth}(\text{const } c) = 0 $$

$$ \text{termDepth}(\text{henkin } c) = 0 $$

$$ \text{termDepth}(\text{zero}) = 0 $$

$$ \text{termDepth}(\text{succ } t) = \text{termDepth}(t) + 1 $$

$$ \text{termDepth}(\text{add } t_1\ t_2) = \max(\text{termDepth}(t_1), \text{termDepth}(t_2)) + 1 $$

$$ \text{termDepth}(\text{mul } t_1\ t_2) = \max(\text{termDepth}(t_1), \text{termDepth}(t_2)) + 1 $$

$$ \text{termDepth}(\text{exp } t_1\ t_2) = \max(\text{termDepth}(t_1), \text{termDepth}(t_2)) + 1 $$

$$ \text{termDepth}(\text{pred } t) = \text{termDepth}(t) + 1 $$

Intuición y necesidad

Mientras que `termSize` mide el número total de nodos del árbol sintáctico (proporcionando combustible para el parser), `termDepth` mide la profundidad de anidamiento de constructores. Esta distinción es crucial para la Fase 1c de la Gödelización: durante la codificación de términos en el lenguaje objeto, ciertos esquemas de representación requieren acotar la profundidad para evitar dependencias circulares en la autorreferencia. La profundidad proporciona una medida alternativa de complejidad sintáctica que resulta más natural para razonar sobre la estructura jerárquica de los códigos. Los casos base (variables, constantes, Henkin, zero) tienen profundidad cero por carecer de subcomponentes, mientras que los constructores unarios incrementan la profundidad en uno y los binarios toman el máximo de las profundidades de sus subárboles más uno.

Función `codeLen`: Longitud de un código

Se define localmente para la Fase 1c la longitud de un código `Code s`, que cuenta el número de elementos en la estructura inductiva:

$$ \text{codeLen} : \text{Code } s \to \mathbb{N} $$

$$ \text{codeLen}(\text{nil}) = 0 $$

$$ \text{codeLen}(\text{cons } x\ xs) = \text{codeLen}(xs) + 1 $$

Intuición y necesidad

Esta función auxiliar proporciona una métrica sobre los códigos que permite razonar sobre su tamaño. A diferencia de la proyección `toList` (que expone la lista subyacente), `codeLen` opera directamente sobre la estructura inductiva de `Code s` sin romper la abstracción estratificada. Es necesaria para establecer cotas inferiores sobre la longitud de los códigos, un requisito técnico para demostrar que existen códigos arbitrariamente largos en cualquier estrato, lo cual es fundamental para los argumentos de diagonalización en la Gödelización.

Función `replicateCode`: Construcción de códigos homogéneos

Se define una función auxiliar que genera un código compuesto exclusivamente por ceros repetidos una cantidad determinada de veces:

$$ \text{replicateCode} : \mathbb{N} \to \text{Code } s $$

$$ \text{replicateCode}(0) = \text{nil} $$

$$ \text{replicateCode}(k+1) = \text{cons}(0, \text{replicateCode}(k)) $$

Intuición y necesidad

`replicateCode` es una herramienta constructiva para generar códigos de longitud arbitraria dentro de cualquier estrato s. Al rellenar con ceros (un valor numérico trivial), se obtienen códigos sintácticamente válidos cuya estructura es completamente predecible. Esta función es el bloque de construcción elemental para el lema de existencia de códigos arbitrariamente largos, que a su vez es indispensable para demostrar que el lenguaje objeto contiene suficientes códigos para representar todas las fórmulas y términos sin colapsar por falta de espacio sintáctico.

Lema `codeLen_replicateCode`

Establece que la longitud del código generado por `replicateCode k` es exactamente k:

$$ \forall s \in \mathbb{N},\ \forall k \in \mathbb{N},\ \text{codeLen}(\text{replicateCode}_{(s:=s)}\ k) = k $$

Intuición y necesidad

Este lema de corrección verifica que `replicateCode` se comporta según lo esperado: genera un código con exactamente k nodos `cons`. La demostración procede por inducción sobre k. El caso base ($k = 0$) es trivial pues `replicateCode 0 = nil` y `codeLen nil = 0`. En el paso inductivo, se despliegan las definiciones de `replicateCode` y `codeLen`, y se aplica la hipótesis de inducción para obtener que la longitud de la cola es k, por lo que la longitud total es $k + 1$. Esta propiedad es esencial para el lema de existencia que sigue, pues garantiza que podemos construir códigos de cualquier longitud deseada.

Lema `exists_code_of_length_gt`: Existencia de códigos arbitrariamente largos

Demuestra que para cualquier número natural n y cualquier estrato s, existe un código en el estrato $s+1$ cuya longitud es estrictamente mayor que n:

$$ \forall s, n \in \mathbb{N},\ \exists c \in \text{Code}(s+1),\ \text{codeLen}(c) > n $$

Intuición y necesidad

Este lema de existencia es técnicamente fundamental para la Fase 1c de la Gödelización. En el proceso de codificación de fórmulas y términos dentro del lenguaje objeto, es necesario garantizar que siempre hay suficientes códigos disponibles para representar nuevas construcciones sintácticas sin agotar el espacio de codificación. La prueba es constructiva: se toma $c = \text{replicateCode}(n+1)$, que por el lema anterior tiene longitud exactamente $n+1$, la cual es estrictamente mayor que n por aritmética elemental. La elección del estrato $s+1$ (en lugar de s) es deliberada y refleja la disciplina de estratificación: los códigos que representan fórmulas del nivel

s típicamente residen en el nivel $s+1$, manteniendo la jerarquía necesaria para evitar paradojas autorreferenciales.

namespace SPA

/-- Profundidad máxima de anidamiento de constructores en un término -/

def termDepth {s : Nat} : ArithTerm s → Nat

| ArithTerm.var _ => 0

| ArithTerm.const _ => 0

| ArithTerm.henkin _ => 0

| ArithTerm.zero => 0

| ArithTerm.succ t => termDepth t + 1

| ArithTerm.add t1 t2 => max (termDepth t1) (termDepth t2) + 1

| ArithTerm.mul t1 t2 => max (termDepth t1) (termDepth t2) + 1

| ArithTerm.exp t1 t2 => max (termDepth t1) (termDepth t2) + 1

| ArithTerm.pred t => termDepth t + 1

/-- Longitud de un código (definida localmente para la Fase 1c) -/

def codeLen {s : Nat} : Code s → Nat

| Code.nil => 0

| Code.cons _ xs => codeLen xs + 1

/-- Función auxiliar: construye un código con k veces `cons 0` anidado -/

def replicateCode {s : Nat} (k : Nat) : Code s :=

```
  match k with
  | 0 => Code.nil
  | m+1 => Code.cons 0 (replicateCode m)

/-- La longitud de replicateCode k es exactamente k -/
lemma codeLen_replicateCode {s : Nat} (k : Nat) : codeLen (replicateCode (s:=s) k) = k
:= by
  induction' k with m ih
  · rfl
  · simp [replicateCode, codeLen, ih]

/-- Para cualquier n, existe un código de nivel s+1 con codeLen > n -/
lemma exists_code_of_length_gt {s n : Nat} : ∃ (c : Code (s+1)), codeLen c > n := by
  use replicateCode (n+1)
  rw [codeLen_replicateCode (n+1)]
  omega
```

Elevación de constantes numéricas entre estratos: `liftNatS`

Se define la función de elevación que transporta un numeral sintáctico del estrato s al estrato inmediatamente superior $s+1$:

$$ \text{liftNatS} : \text{NatS } s \to \text{NatS}(s + 1) $$

con las ecuaciones recursivas:

$$ \text{liftNatS}(\text{zero}) = \text{zero} $$

$$ \text{liftNatS}(\text{succ } n) = \text{succ}(\text{liftNatS } n) $$

Intuición y necesidad

En la aritmética estratificada, los numerales `NatS s` están indexados por un parámetro fantasma s que restringe su uso a un nivel sintáctico específico. Sin embargo, las constantes numéricas son inherentemente transportables entre estratos, pues su estructura interna (cero y sucesor) no depende del nivel. `liftNatS` formaliza esta transportabilidad elevando un numeral a cualquier estrato superior preservando su valor numérico. Esta función es el bloque elemental sobre el cual se construye la elevación de términos y fórmulas completas.

Elevación de términos entre estratos: `liftTerm`

Se define el levantamiento estructural de un término aritmético completo del estrato s al estrato $s+1$:

$$ \text{liftTerm} : \text{ArithTerm } s \to \text{ArithTerm}(s + 1) $$

definido recursivamente sobre la estructura del término. Las variables, constantes de Henkin y el cero se preservan idénticamente; las constantes numéricas se elevan mediante `liftNatS`; los constructores compuestos (sucesor, suma, producto, exponenciación, predecesor) se elevan aplicando `liftTerm` recursivamente a sus subcomponentes:

$$ \text{liftTerm}(\text{var } n) = \text{var } n $$

$$ \text{liftTerm}(\text{const } c) = \text{const}(\text{liftNatS } c) $$

$$ \text{liftTerm}(\text{henkin } c) = \text{henkin } c $$

$$ \text{liftTerm}(\text{zero}) = \text{zero} $$

$$ \text{liftTerm}(\text{succ } t) = \text{succ}(\text{liftTerm } t) $$

$$ \text{liftTerm}(\text{add } t_1\ t_2) = \text{add}(\text{liftTerm } t_1)(\text{liftTerm } t_2) $$

y análogamente para `mul`, `exp` y `pred`.

Intuición y necesidad

La elevación de términos es una operación fundamental en la Fase 1C, dedicada a la no representabilidad de la inyección con desplazamiento de niveles. En sistemas estratificados, una fórmula del nivel s puede necesitar referirse a términos del nivel s dentro de operadores que exigen el nivel $s+1$ (como los operadores de demostrabilidad y verdad). `liftTerm` provee el mecanismo sintáctico para trasladar términos respetando la disciplina de estratificación: el término elevado es estructuralmente idéntico al original pero habita el estrato superior, permitiendo su uso en contextos que requieren el nivel $s+1$.

Elevación de fórmulas entre estratos: `liftForm`

Se extiende la elevación al nivel de las fórmulas lógicas, transportando una fórmula completa del estrato s al estrato $s+1$:

$$ \text{liftForm} : \text{ArithForm } s \to \text{ArithForm}(s + 1) $$

La elevación actúa recursivamente sobre la estructura de la fórmula. Los casos atómicos (`bottom`, igualdad, pertenencia a listas) elevan sus subtérminos mediante `liftTerm`. Los conectivos lógicos (implicación, negación) y el cuantificador universal elevan sus subfórmulas recursivamente. El tratamiento de los operadores estratificados `provable_op` y `true_op` merece especial atención: si una fórmula contiene una ocurrencia de `provable_op` con testigo de igualdad $h : s = s_{\text{lower}} + 1$, al elevar la fórmula al nivel $s+1$, el nuevo testigo debe reflejar que $(s+1) = (s_{\text{lower}} + 1) + 1$, lo cual se obtiene reescribiendo con h:

$$ \text{liftForm}(\text{provable_op}_{(s_{\text{lower}}:=sl)}\ h\ t) = \text{provable_op}_{(s_{\text{lower}}:=sl+1)}\ (\text{por reescritura de } h)\ (\text{liftTerm } t) $$

Intuición y necesidad

`liftForm` completa el aparato de elevación sintáctica, permitiendo que fórmulas completas migren al estrato superior. Esta operación es indispensable para analizar la no representabilidad de la inyección: si existiera una fórmula $\varphi(x)$ en el nivel s que representara la propiedad "el código x corresponde a una fórmula del nivel $s-1$", podríamos elevarla a $\text{liftForm}(\varphi)$ en el nivel $s+1$ y obtener una contradicción por diagonalización. La corrección del desplazamiento de los índices de estrato en `provable_op` y `true_op` asegura que la jerarquía se preserva consistentemente durante la elevación.

Lema `isClosed_liftTerm`: Preservación de la clausura bajo elevación

Establece que la propiedad de ser un término cerrado (sin variables libres) es invariante bajo la operación de elevación de estrato:

$$ \forall s \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \text{isClosed}(\text{liftTerm } t) = \text{isClosed}(t) $$

Intuición y necesidad

Un término es cerrado si no contiene variables libres; esta propiedad es puramente estructural y no depende del estrato en el que reside el término. La demostración procede por inducción estructural sobre t: los casos base (variables, constantes, Henkin, zero) son inmediatos pues `liftTerm` no altera la presencia de variables; en los casos recursivos (sucesor, suma, etc.), se despliega la definición de `liftTerm` y de `isClosed`, y se aplica la hipótesis de inducción a cada subcomponente. Este lema es necesario para justificar la construcción de `liftTermClosed`, que eleva términos garantizando que la clausura se preserva, una propiedad esencial para mantener la corrección sintáctica al trasladar términos entre estratos en contextos deductivos.

Elevación de términos cerrados: `liftTermClosed`

Se define la elevación específica para el subtipo de términos cerrados, produciendo un término cerrado en el estrato superior:

$$ \text{liftTermClosed} : \text{TermClosed } s \to \text{TermClosed}(s + 1) $$

donde $\text{TermClosed } s$ es el subtipo de $\text{ArithTerm } s$ que satisface el predicado `isClosed`. La construcción toma un término cerrado t con su prueba de clausura, aplica `liftTerm` al valor subyacente y utiliza el lema `isClosed_liftTerm` para demostrar que el término elevado también es cerrado.

Intuición y necesidad

En muchos contextos deductivos, solo los términos cerrados son admisibles para sustitución e instanciación. `liftTermClosed` proporciona una interfaz tipada que garantiza estáticamente la preservación de la clausura durante la elevación, evitando la necesidad de reverificar la propiedad después de cada traslado. Esta construcción es particularmente relevante para la manipulación de testigos de Henkin y fórmulas existenciales, donde los términos que se sustituyen deben ser cerrados para preservar la corrección lógica.

--
==========
==

-- FASE 1C: NO REPRESENTABILIDAD DE LA INYECCIÓN CON DESPLAZAMIENTO DE NIVELES

--
==========
==

```
/-- Elevación de constantes numéricas de estrato s al estrato s+1 -/
def liftNatS {s : Nat} : NatS s → NatS (s + 1)
  | NatS.zero => NatS.zero
  | NatS.succ n => NatS.succ (liftNatS n)

/-- Elevación de términos de estrato s al estrato s+1 -/
def liftTerm {s : Nat} : ArithTerm s → ArithTerm (s + 1)
  | ArithTerm.var n => ArithTerm.var n
  | ArithTerm.const c => ArithTerm.const (liftNatS c)
  | ArithTerm.henkin c => ArithTerm.henkin c
  | ArithTerm.zero => ArithTerm.zero
  | ArithTerm.succ t => ArithTerm.succ (liftTerm t)
  | ArithTerm.add t1 t2 => ArithTerm.add (liftTerm t1) (liftTerm t2)
  | ArithTerm.mul t1 t2 => ArithTerm.mul (liftTerm t1) (liftTerm t2)
  | ArithTerm.exp t1 t2 => ArithTerm.exp (liftTerm t1) (liftTerm t2)
  | ArithTerm.pred t => ArithTerm.pred (liftTerm t)

/-- Elevación de fórmulas de estrato s al estrato s+1 -/
def liftForm {s : Nat} : ArithForm s → ArithForm (s + 1)
  | ArithForm.bottom => ArithForm.bottom
  | ArithForm.eq t1 t2 => ArithForm.eq (liftTerm t1) (liftTerm t2)
  | ArithForm.impl f1 f2 => ArithForm.impl (liftForm f1) (liftForm f2)
  | ArithForm.neg f => ArithForm.neg (liftForm f)
  | ArithForm.forall_form n f => ArithForm.forall_form n (liftForm f)
  | ArithForm.provable_op (s_lower := sl) h t =>
    ArithForm.provable_op (s_lower := sl + 1) (by rw [h]) (liftTerm t)
  | ArithForm.true_op (s_lower := sl) h t =>
```

```
   ArithForm.true_op (s_lower := sl + 1) (by rw [h]) (liftTerm t)
  | ArithForm.list_nth_eq t1 t2 t3 => ArithForm.list_nth_eq (liftTerm t1) (liftTerm t2)
(liftTerm t3)


/-- Lema de preservación de la clausura de términos bajo elevación -/
lemma isClosed_liftTerm {s : Nat} (t : ArithTerm s) :
   isClosed (liftTerm t) = isClosed t := by
  induction t with
  | var n => rfl
  | const c => rfl
  | henkin c => rfl
  | zero => rfl
  | succ t' ih =>
    dsimp [liftTerm, isClosed]
    exact ih
  | add t1 t2 ih1 ih2 =>
    dsimp [liftTerm, isClosed]
    rw [ih1, ih2]
  | mul t1 t2 ih1 ih2 =>
    dsimp [liftTerm, isClosed]
    rw [ih1, ih2]
  | exp t1 t2 ih1 ih2 =>
    dsimp [liftTerm, isClosed]
    rw [ih1, ih2]
  | pred t' ih =>
    dsimp [liftTerm, isClosed]
    exact ih
```

```
/-- Elevación de términos cerrados del estrato s al estrato s+1 -/
def liftTermClosed {s : Nat} (t : TermClosed s) : TermClosed (s + 1) :=
  ⟨liftTerm t.val, by
    rw [isClosed_liftTerm]
    exact t.property⟩
```

Lema `liftTerm_substTerm`: Conmutación de la elevación con la sustitución de términos

Establece que la operación de elevación de estrato conmuta con la sustitución sintáctica de una variable por un término. Formalmente, para todo término t, variable v y término de sustitución u en el estrato s:

$$ \forall s \in \mathbb{N},\ \forall t, u \in \text{ArithTerm } s,\ \forall v \in \mathbb{N},\ \text{liftTerm}(\text{substTerm}(v, u, t)) = \text{substTerm}(v, \text{liftTerm}(u), \text{liftTerm}(t)) $$

Intuición y necesidad

Este lema de conmutación captura una propiedad algebraica fundamental: elevar un término después de sustituir es equivalente a sustituir después de elevar tanto el término como el término de sustitución. La demostración procede por inducción estructural sobre t. En el caso base de variable (`var n`), se analiza si $n = v$ mediante un desdoblamiento condicional: si la variable coincide con la variable de sustitución, ambos lados se reducen a $\text{liftTerm}(u)$; en caso contrario, ambos lados preservan la variable intacta. Los casos de constantes, Henkin y cero son triviales pues no contienen variables. Para los constructores compuestos (sucesor, suma, producto, exponenciación, predecesor), la prueba despliega las definiciones de `substTerm` y `liftTerm`, y aplica las hipótesis de inducción a los subcomponentes, recomponiendo la igualdad estructuralmente. Esta propiedad de conmutación es esencial en la Fase 1C porque garantiza que el levantamiento de estrato se comporta functorialmente respecto a la sustitución, una condición necesaria para que los diagramas sintácticos de la Gödelización sean coherentes.

Lema `liftForm_substForm`: Conmutación de la elevación con la sustitución de fórmulas

Extiende la propiedad de conmutación al nivel de las fórmulas lógicas. Para toda fórmula φ, variable v y término de sustitución u en el estrato s:

$$ \forall s \in \mathbb{N},\ \forall \varphi \in \text{ArithForm } s,\ \forall v \in \mathbb{N},\ \forall u \in \text{ArithTerm } s,\ \text{liftForm}(\text{substForm}(v, u, \varphi)) = \text{substForm}(v, \text{liftTerm}(u), \text{liftForm}(\varphi)) $$

Intuición y necesidad

Este lema generaliza la conmutación a toda la sintaxis de fórmulas, cubriendo todos los constructores lógicos del lenguaje. La demostración procede por inducción sobre φ con generalización de las variables v y u. Los casos atómicos (`bottom`, igualdad, pertenencia a listas) se reducen al lema anterior `liftTerm_substTerm` aplicado a sus subtérminos. Los conectivos proposicionales (implicación, negación) aplican directamente las hipótesis de inducción. El caso del cuantificador universal (`forall_form n f`) requiere un análisis adicional: la sustitución solo afecta a f si la variable ligada n difiere de v; cuando $n = v$, la sustitución no modifica la fórmula y la igualdad se cumple trivialmente; cuando $n \neq v$, se aplica la hipótesis de inducción. Los operadores estratificados `provable_op` y `true_op` se reducen nuevamente a `liftTerm_substTerm` pues solo contienen un término como subcomponente sintáctico. La validez de esta conmutación es crucial para la maquinaria de Gödelización: asegura que las operaciones de codificación y descodificación respetan la estructura de sustitución, lo cual es indispensable para demostrar los lemas de diagonalización y los teoremas de punto fijo que subyacen a los resultados de incompletitud.

```
/-- Conmutación de la elevación con la sustitución ordinaria de términos -/
lemma liftTerm_substTerm {s : Nat} (t : ArithTerm s) (v : Nat) (u : ArithTerm s) :
  liftTerm (substTerm v u t) = substTerm v (liftTerm u) (liftTerm t) := by
```

```
  induction t with
  | var n =>
    dsimp [substTerm, liftTerm]
    split_ifs with h_vn
    · rfl
    · rfl
  | const c => rfl
  | henkin c => rfl
  | zero => rfl
  | succ t' ih =>
    dsimp [substTerm, liftTerm]
    rw [ih]
  | add t1 t2 ih1 ih2 =>
    dsimp [substTerm, liftTerm]
    rw [ih1, ih2]
  | mul t1 t2 ih1 ih2 =>
    dsimp [substTerm, liftTerm]
    rw [ih1, ih2]
  | exp t1 t2 ih1 ih2 =>
    dsimp [substTerm, liftTerm]
    rw [ih1, ih2]
  | pred t' ih =>
    dsimp [substTerm, liftTerm]
    rw [ih]

/-- Conmutación de la elevación con la sustitución de fórmulas -/
lemma liftForm_substForm {s : Nat} (φ : ArithForm s) (v : Nat) (u : ArithTerm s) :
    liftForm (substForm v u φ) = substForm v (liftTerm u) (liftForm φ) := by
```

```
induction φ generalizing v u with
| bottom => rfl
| eq t1 t2 =>
  dsimp [substForm, liftForm]
  rw [liftTerm_substTerm t1, liftTerm_substTerm t2]
| impl f1 f2 ih1 ih2 =>
  dsimp [substForm, liftForm]
  rw [ih1, ih2]
| neg f ih =>
  dsimp [substForm, liftForm]
  rw [ih]
| forall_form n f ih =>
  dsimp [substForm, liftForm]
  split_ifs with h_vn
  · rfl
  · dsimp [liftForm]
    rw [ih]
| provable_op h t =>
  dsimp [substForm, liftForm]
  rw [liftTerm_substTerm]
| true_op h t =>
  dsimp [substForm, liftForm]
  rw [liftTerm_substTerm]
| list_nth_eq t1 t2 t3 =>
  dsimp [substForm, liftForm]
  rw [liftTerm_substTerm t1, liftTerm_substTerm t2, liftTerm_substTerm t3]
```

Función `natToTerm`: Construcción del numeral de Peano

Se define la función que convierte un número natural del metalenguaje en su representación como numeral sintáctico dentro del lenguaje objeto, utilizando el constructor `zero` y aplicaciones iteradas del constructor `succ`:

$$ \text{natToTerm} : \mathbb{N} \to \text{ArithTerm } s $$

$$ \text{natToTerm}(0) = \text{zero} $$

$$ \text{natToTerm}(k+1) = \text{succ}(\text{natToTerm}(k)) $$

Intuición y necesidad

Esta función establece el puente fundamental entre los números naturales del metalenguaje (donde razonamos sobre codificaciones y demostraciones) y los numerales del lenguaje objeto (donde residen las fórmulas y los términos de la teoría). El numeral $\text{natToTerm}(n)$ corresponde al término $\text{succ}^n(\text{zero})$, es decir, la representación canónica de n como numeral de Peano. La parametrización por el estrato s es un parámetro fantasma que permite insertar estos numerales en cualquier nivel sintáctico sin alterar su estructura interna. Esta función es el ingrediente básico para la codificación interna: los códigos de Gödel (que son números naturales) deben poder expresarse como términos del lenguaje para que las fórmulas puedan referirse a ellos.

Lema `isClosed_natToTerm`: Clausura de los numerales de Peano

Establece que todo numeral de Peano construido mediante `natToTerm` es un término cerrado, es decir, no contiene variables libres:

$$ \forall s \in \mathbb{N},\ \forall n \in \mathbb{N},\ \text{isClosed}(\text{natToTerm}_{(s:=s)}\ n) = \text{true} $$

Intuición y necesidad

La clausura de los numerales es una propiedad sintáctica elemental pero indispensable: dado que los numerales se construyen exclusivamente a partir de `zero` y `succ` sin involucrar variables, son inherentemente cerrados. La demostración procede por inducción sobre n: el caso base ($n=0$) es trivial pues `zero` es cerrado; en el paso inductivo, se despliega la definición de `natToTerm` y de `isClosed`, reduciendo la verificación a la hipótesis de inducción. Esta propiedad permite empaquetar cualquier numeral como un `TermClosed s`, el subtipo de términos que garantiza estáticamente la ausencia de variables libres, condición necesaria para la sustitución segura en contextos cuantificados.

Función `internalCode`: Codificación interna de un código como numeral cerrado

Se define la función que toma un código del estrato $s+1$ y lo convierte en un término cerrado del estrato s:

$$ \text{internalCode} : \text{Code}(s+1) \to \text{TermClosed } s $$

$$ \text{internalCode}(c) = \langle \text{natToTerm}(\text{encode}(c)),\ \text{isClosed_natToTerm}(\text{encode}(c)) \rangle $$

donde encode es la codificación provista por la instancia `Encodable` de `Code (s+1)`.

Intuición y necesidad

`internalCode` materializa el paso crucial de la Gödelización: transforma un código abstracto (que vive en el nivel $s+1$) en un numeral concreto del lenguaje objeto (que vive en el nivel s). El desplazamiento de niveles es deliberado y refleja la jerarquía estratificada: los códigos de las fórmulas del nivel s residen en el nivel $s+1$, pero cuando una fórmula del nivel s necesita hablar sobre códigos, debe hacerlo a través de numerales que habitan su mismo nivel. La función utiliza `Encodable.encode` para

obtener el número natural asociado al código, `natToTerm` para convertirlo en numeral sintáctico, y `isClosed_natToTerm` para certificar que el resultado es cerrado. El resultado se empaqueta en el subtipo `TermClosed s`, garantizando estáticamente que puede usarse en sustituciones sin riesgo de captura de variables.

Lema `internalCode_injective`: Inyectividad de la codificación interna

Demuestra que la función `internalCode` es inyectiva: códigos distintos del nivel $s+1$ producen numerales cerrados distintos en el nivel s:

$$ \forall s \in \mathbb{N},\ \forall c_1, c_2 \in \text{Code}(s+1),\ \text{internalCode}(c_1) = \text{internalCode}(c_2) \implies c_1 = c_2 $$

Intuición y necesidad

La inyectividad de la codificación interna es una propiedad semántica fundamental sin la cual la Gödelización colapsaría: si dos códigos distintos pudieran mapearse al mismo numeral, el lenguaje objeto no podría distinguir entre ellos y la representabilidad de funciones sobre códigos perdería sentido. La demostración descompone la igualdad de términos cerrados (que son pares dependientes) en la igualdad de sus componentes subyacentes mediante `injection`. Se requiere entonces probar que `natToTerm` es inyectiva como función de $\mathbb{N}$ a `ArithTerm s`. Esta sub-demostración procede por inducción anidada: se demuestra que si $\text{natToTerm}(n) = \text{natToTerm}(m)$, entonces $n = m$, analizando los casos de construcción de ambos numerales. Con la inyectividad de `natToTerm` establecida, se obtiene $\text{encode}(c_1) = \text{encode}(c_2)$, y la inyectividad de `Encodable.encode` (provista por la instancia `Encodable`) permite concluir $c_1 = c_2$. Este lema garantiza que la representación interna de códigos es fiel.

Predicado `IsRepresentable`: Definición sintáctica de representabilidad en Δ

Se define formalmente qué significa que una función f que mapea códigos del nivel $s+1$ a términos cerrados del nivel s sea representable dentro de un conjunto de fórmulas Δ:

$$ \text{IsRepresentable} : \Delta \subseteq \text{ArithForm } s,\ f : \text{Code}(s+1) \to \text{TermClosed } s \vdash \text{Prop} $$

La definición exige la existencia de una fórmula $\varphi \in \text{ArithForm } s$ que satisface simultáneamente tres condiciones estructurales y dos condiciones funcionales:

Condiciones estructurales sobre las variables libres de φ:

1. La variable v_0 ocurre libre en φ: $\neg \text{NotFreeForm}(0, \varphi)$

2. La variable v_1 ocurre libre en φ: $\neg \text{NotFreeForm}(1, \varphi)$

3. Ninguna otra variable ocurre libre en φ: $\forall v \in \mathbb{N},\ v \neq 0 \land v \neq 1 \implies \text{NotFreeForm}(v, \varphi)$

Condiciones funcionales sobre el comportamiento deductivo en Δ:

(1) **Completitud del gráfico:** Para cada código $c \in \text{Code}(s+1)$, la instancia de φ obtenida al sustituir v_0 por el numeral que codifica c y v_1 por el término $f(c)$ pertenece a Δ:

$$ \forall c \in \text{Code}(s+1),\ \varphi[v_0 := \text{internalCode}(c),\ v_1 := f(c)] \in \Delta $$

(2) **Funcionalidad:** Si para algún código c y algún término cerrado t, la instancia $\varphi[v_0 := \text{internalCode}(c),\ v_1 := t]$ pertenece a Δ, entonces la igualdad $f(c) = t$ también pertenece a Δ:

$$ \forall c \in \text{Code}(s+1),\ \forall t \in \text{TermClosed } s,\ \varphi[v_0 := \text{internalCode}(c),\ v_1 := t] \in \Delta \implies (f(c) = t) \in \Delta $$

Intuición y necesidad

Esta definición refinada y corregida de representabilidad constituye el núcleo conceptual de la Fase 1C. Captura sintácticamente la noción de que una función sobre códigos puede ser "definida" o "representada" dentro de la teoría Δ mediante una fórmula con exactamente dos variables libres. La condición (1) exige que Δ contenga todas las instancias del gráfico de f, es decir, que Δ "reconozca" los pares entrada-salida correctos. La condición (2) exige funcionalidad: si Δ contiene un par (c, t) según φ, entonces debe poder demostrar que t coincide con $f(c)$. La restricción a exactamente dos variables libres (v_0 y v_1) garantiza que φ define una relación binaria sin parámetros ocultos. Esta definición es puramente sintáctica y no apela a nociones semánticas externas, lo cual es indispensable para que los argumentos de incompletitud se desarrollen íntegramente dentro del sistema formal. El desplazamiento de niveles (f toma códigos de $s+1$ y devuelve términos de s) refleja la estratificación esencial para evitar la autorreferencia paradójica.

```
--
============================================================
==

-- 1c.2: DEFINICIÓN SINTÁCTICA DE IsRepresentable (REFINADA Y CORREGIDA)

--
============================================================
==


/-- Función auxiliar: construye el numeral de Peano succ^k (zero) -/

def natToTerm {s : Nat} : Nat → ArithTerm s

 | 0 => ArithTerm.zero

 | k+1 => ArithTerm.succ (natToTerm k)


/-- El numeral de Peano es siempre cerrado -/

lemma isClosed_natToTerm {s : Nat} (n : Nat) : isClosed (natToTerm (s := s) n) = true
:= by

 induction' n with m ih

 · rfl

 · dsimp [natToTerm, isClosed]
```

```
    exact ih

/-- Codificación interna de un código de nivel s+1 como numeral de Peano de nivel s -/
def internalCode {s : Nat} (c : Code (s+1)) : TermClosed s :=
  let n := Encodable.encode c
  ⟨natToTerm n, isClosed_natToTerm n⟩

/-- La codificación interna es inyectiva -/

lemma internalCode_injective {s : Nat} : Function.Injective (internalCode (s := s)) := by
  intro c1 c2 h
  unfold internalCode at h
  injection h with h_term
  have h_len : Encodable.encode c1 = Encodable.encode c2 := by
    -- natToTerm es inyectiva
    have h_nat_inj : ∀ {n m : Nat}, natToTerm (s := s) n = natToTerm m → n = m := by
      intro n m h_eq
      induction n generalizing m with
      | zero =>
        cases m
        · rfl
        · dsimp [natToTerm] at h_eq
          injection h_eq
      | succ k ih =>
        cases m
        · dsimp [natToTerm] at h_eq
          injection h_eq
        · dsimp [natToTerm] at h_eq
```

```
      injection h_eq with h_eq'
      rw [ih h_eq']
  exact h_nat_inj h_term
 exact Encodable.encode_injective h_len


/--
  Definición puramente sintáctica de representabilidad en Δ.
  Una función f : Code (s+1) → TermClosed s es representable si existe
  una fórmula φ con exactamente dos variables libres (v₀, v₁) tal que
  para todo código c, φ define funcionalmente a f(c) dentro de Δ.
-/
def IsRepresentable {s : Nat} (Δ : Set (ArithForm s)) (f : Code (s+1) → TermClosed s) :
Prop :=
 ∃ (φ : ArithForm s)
  -- v₀ y v₁ son exactamente las variables libres de φ
  (_h_v0 : ¬ NotFreeForm 0 φ)
  (_h_v1 : ¬ NotFreeForm 1 φ)
  (_h_only : ∀ (v : Nat), v ≠ 0 → v ≠ 1 → NotFreeForm v φ),
  -- (1) Gráfico: para cada código c, φ(internalCode c, f(c)) ∈ Δ
  (∀ (c : Code (s+1)),
   let φ_c := substForm 1 (f c).val (substForm 0 (internalCode c).val φ)
   φ_c ∈ Δ) ∧
  -- (2) Funcionalidad: si φ(internalCode c, t) ∈ Δ, entonces f(c) = t es demostrable en
Δ
  (∀ (c : Code (s+1)) (t : TermClosed s),
   let φ_ct := substForm 1 t.val (substForm 0 (internalCode c).val φ)
   φ_ct ∈ Δ → ArithForm.eq (f c).val t.val ∈ Δ)
```

Fórmula diagonal `diagonalFormula`

Se define la fórmula diagonal $\psi(x)$ en el estrato $s+1$ a partir de una fórmula φ' del mismo estrato. La construcción formal es:

$$ \text{diagonalFormula}(\varphi') := \forall v_1\ (\varphi'(v_1) \to \neg\ \text{provable_op}(v_1, x)) $$

donde la variable ligada es v_1 y la variable libre implícita es v_0, que ocupa la posición de x en la definición. Explícitamente, utilizando los constructores del lenguaje:

$$ \text{diagonalFormula}(\varphi') = \text{forall_form}(1,\ \text{impl}(\varphi',\ \text{neg}(\text{provable_op}_{(s_{\text{lower}}:=s)}\ \text{rfl}\ (\text{var } 1)))) $$

Intuición y necesidad

Esta fórmula es la pieza central del argumento diagonal de Gödel adaptado al sistema estratificado SPA. La intuición es la siguiente: φ' es una fórmula con variables libres v_0 y v_1 que pretende representar una relación binaria. La fórmula diagonal $\psi(v_0)$ expresa: "para todo v_1, si $\varphi'(v_0, v_1)$ se cumple, entonces v_1 no es una prueba de v_0". En otras palabras, $\psi(v_0)$ afirma que ningún v_1 que satisfaga la relación φ' con v_0 puede ser una prueba de v_0.

El operador `provable_op` con índice de estrato inferior $s_{\text{lower}} := s$ y testigo `rfl` (que certifica $s+1 = s + 1$) aplicado a la variable v_1 codifica la noción de que v_1 es una prueba en el nivel s. La variable ligada 1 recorre todos los posibles testigos y, y la implicación afirma que si $\varphi'(x, y)$ se cumple, entonces y no demuestra x. Esta construcción es el análogo estratificado de la fórmula clásica $\forall y\ (\varphi(x, y) \to \neg\ \text{Prov}(y, x))$ utilizada en la prueba original de Gödel, pero respetando la jerarquía de niveles para evitar la autorreferencia irrestricta.

Conjunto inicial Γ_0 de nivel $s+1$

Se define el conjunto Γ_0 como la base axiomática del estrato $s+1$, construida a partir de un conjunto maximal Δ de fórmulas del estrato s y de los axiomas recursivos de la aritmética:

$$ \Gamma_0(\Delta) := \{\ \varphi' \mid \exists \varphi \in \Delta,\ \varphi' = \text{liftForm}(\varphi)\ \} \cup \{\ \varphi' \mid \exists x \in \text{ArithTerm}(s+1),\ \varphi' = \text{add_zero_axiom}(x)\ \} \cup \{\ \varphi' \mid \exists x, y \in \text{ArithTerm}(s+1),\ \varphi' = \text{add_succ_axiom}(x, y)\ \} $$

Intuición y necesidad

Γ_0 constituye el punto de partida para la construcción de la secuencia de Henkin en el nivel $s+1$, en el espíritu del Lema de Lindenbaum aplicado estratificadamente. El conjunto se compone de tres familias:

1. **Elevación de Δ:** Todas las fórmulas del conjunto maximal Δ del nivel s se elevan al nivel $s+1$ mediante `liftForm`. Esto asegura que la teoría del nivel superior herede todo el contenido deductivo del nivel inferior, manteniendo la coherencia vertical entre estratos. Si Δ es maximalmente consistente en el nivel s, su elevación proporciona una base sólida para la extensión en el nivel $s+1$.

2. **Axiomas de la suma:** Se incluyen todas las instancias de los axiomas recursivos de la suma (`add_zero_axiom` y `add_succ_axiom`) para los términos del nivel $s+1$.

Estos axiomas dotan a la teoría del nivel superior de la estructura aritmética básica necesaria para razonar sobre numerales y codificaciones.

La construcción de Γ_0 como unión de estos conjuntos refleja la arquitectura estratificada del sistema: el conocimiento se transfiere hacia arriba mediante `liftForm`, mientras que los axiomas aritméticos se reinstancian en cada nivel para garantizar que las operaciones básicas sobre numerales estén disponibles localmente. Esta base axiomática será posteriormente extendida con testigos de Henkin y otras fórmulas hasta obtener una teoría completa en el nivel $s+1$.

```
/-- Fórmula diagonal ψ(x) en el estrato s+1: ∀ y (φ'(x, y) → ¬ provable_op(y, x)) -/
def diagonalFormula {s : Nat} (φ' : ArithForm (s + 1)) : ArithForm (s + 1) :=
 ArithForm.forall_form 1 (
  ArithForm.impl
   φ'
   (ArithForm.neg (ArithForm.provable_op (s_lower := s) rfl (ArithTerm.var 1)))
 )

/-- Conjunto inicial Γ₀ de nivel s+1 construido a partir del conjunto maximal Δ de nivel s -/
def Gamma0 {s : Nat} (Δ : Set (ArithForm s)) : Set (ArithForm (s + 1)) :=
 { φ' | ∃ φ ∈ Δ, φ' = liftForm φ } ∪
 { φ' | ∃ (x : ArithTerm (s+1)), φ' = add_zero_axiom x } ∪
 { φ' | ∃ (x y : ArithTerm (s+1)), φ' = add_succ_axiom x y }
```

Esquemas axiomáticos de Tarski para el operador de verdad

Se define el conjunto de axiomas de Tarski para el nivel $s+1$, que establecen la correspondencia semántica entre una fórmula del nivel s y el operador de verdad `true_op` aplicado a su código en el nivel $s+1$:

$$ \text{TarskiAxioms}(s) := \{\ \varphi' \mid \exists \theta \in \text{ArithForm } s,\ \varphi' = \text{true_op}(\overline{\text{encode}(\theta)}) \to \text{liftForm}(\theta)\ \} \cup \{\ \varphi' \mid \exists \theta \in \text{ArithForm } s,\ \varphi' = \text{liftForm}(\theta) \to \text{true_op}(\overline{\text{encode}(\theta)})\ \} $$

donde $\overline{\text{encode}(\theta)}$ denota el numeral sintáctico correspondiente al código de θ, construido mediante `NatS_decode` aplicado al resultado de `Encodable.encode`. Formalmente, las dos direcciones del bicondicional de Tarski se postulan como axiomas separados:

Dirección de eliminación (T-out): Si el operador de verdad afirma θ, entonces θ se cumple (elevada al nivel superior):

$$ \forall \theta \in \text{ArithForm } s,\ \text{true_op}(\overline{\text{encode}(\theta)}) \to \text{liftForm}(\theta) \in \text{TarskiAxioms}(s) $$

Dirección de introducción (T-in): Si θ se cumple (elevada), entonces el operador de verdad lo afirma:

$$ \forall \theta \in \text{ArithForm } s,\ \text{liftForm}(\theta) \to \text{true_op}(\overline{\text{encode}(\theta)}) \in \text{TarskiAxioms}(s) $$

Intuición y necesidad

Los axiomas de Tarski capturan la intuición semántica fundamental del predicado de verdad: una fórmula θ es verdadera exactamente cuando θ se cumple. En el sistema estratificado SPA, esta correspondencia debe formularse con cuidado: la fórmula θ pertenece al nivel s, mientras que la afirmación sobre su verdad (`true_op`) pertenece al nivel $s+1$. Los axiomas vinculan estos dos niveles mediante el esquema $T(\overline{\theta}) \leftrightarrow \theta'$, donde θ' es la elevación de θ al nivel $s+1$ mediante `liftForm`. La codificación $\overline{\text{encode}(\theta)}$ convierte la fórmula θ en un numeral que el operador `true_op` puede recibir como argumento. Esta formulación es la adaptación estratificada de la convención-T de Tarski, diseñada para operar dentro de la jerarquía de niveles sin provocar la paradoja del mentiroso, pues el predicado de verdad de un nivel solo se aplica a fórmulas del nivel inmediatamente inferior.

Conjunto inicial Γ_0 para el nivel base ($s = 0$)

Se define el conjunto axiomático de partida para el nivel 1, especializando la construcción general al caso $s = 0$ e incorporando adicionalmente los axiomas de Tarski:

$$ \Gamma_0^{\text{zero}}(\Delta) := \{\ \varphi' \mid \exists \varphi \in \Delta,\ \varphi' = \text{liftForm}(\varphi)\ \} \cup \{\ \varphi' \mid \exists x \in \text{ArithTerm } 1,\ \varphi' = \text{add_zero_axiom}(x)\ \} \cup \{\ \varphi' \mid \exists x, y \in \text{ArithTerm } 1,\ \varphi' = \text{add_succ_axiom}(x, y)\ \} \cup \text{TarskiAxioms}(0) $$

Intuición y necesidad

Γ_0^{zero} es el análogo concreto de Γ_0 para el primer nivel no trivial de la jerarquía. Además de la elevación del conjunto maximal Δ del nivel 0 y de los axiomas recursivos de la suma, incorpora explícitamente los axiomas de Tarski para el nivel 0. Esta inclusión es crucial porque el nivel 1 es el primer estrato donde el operador de verdad `true_op` puede aplicarse (ya que requiere un nivel inferior $s_{\text{lower}} = 0$). Al incluir `TarskiAxioms 0`, la teoría de partida ya contiene las propiedades básicas del predicado de verdad para las fórmulas del nivel 0, estableciendo la base sobre la cual se desarrollará el argumento de diagonalización y no representabilidad específico para $s = 0$.

Infraestructura de Henkin-Lindenbaum para el nivel 1

Se replica la maquinaria completa de construcción de la secuencia de Henkin y el límite de Lindenbaum para el estrato 1, adaptando las definiciones generales a este nivel concreto. Las definiciones incluyen:

`GammaWitnessesList1`: Lista de testigos de Henkin acumulados durante la enumeración de fórmulas existenciales del nivel 1. En cada paso $m+1$, se examina la fórmula correspondiente en la enumeración (`decodeExists (s := 1) m`); si contiene componentes existenciales, se genera una variable fresca w y una constante de Henkin fresca c, y se añade el axioma de testigo:

$$ \text{freshEx} \to \psi[v := \text{henkin}(c)] $$

donde freshEx es la fórmula existencial cerrada $\neg \forall w \neg \psi[v := \text{var } w]$. Las cotas para las variables y constantes frescas se calculan mediante los operadores `max_var_list`, `max_var_form`, `maxHenkinList` y `maxHenkinForm`, asegurando que los nuevos símbolos no colisionen con los ya utilizados.

`GammaSeq1`: Secuencia creciente de conjuntos de fórmulas del nivel 1, indexada por números naturales, que en cada paso añade el axioma de testigo correspondiente si la fórmula enumerada contiene componentes existenciales.

`GammaLimit1`: Límite de la secuencia de Henkin, definido como la unión de todos los conjuntos $\text{GammaSeq1}(\Gamma, n)$ para $n \in \mathbb{N}$:

$$ \text{GammaLimit1}(\Gamma) := \bigcup_{n \in \mathbb{N}} \text{GammaSeq1}(\Gamma, n) $$

`LindenbaumSeq1`: Secuencia de extensión de Lindenbaum que, partiendo de `GammaLimit1`, procesa cada fórmula del nivel 1 (mediante la enumeración `Encodable.decode`) y decide si añadirla o añadir su negación, preservando la consistencia en cada paso.

`LindenbaumLimit1`: Límite final de Lindenbaum, unión de toda la secuencia, que constituye un conjunto maximalmente consistente de fórmulas del nivel 1.

Intuición y necesidad

Esta infraestructura replica para el nivel 1 el procedimiento estándar de Henkin-Lindenbaum que garantiza la existencia de extensiones completas y existencialmente saturadas de una teoría consistente. La construcción es necesaria para la Fase 1C-α porque el argumento de diagonalización y no representabilidad requiere trabajar en una teoría completa y con testigos para todas las fórmulas existenciales del nivel 1. La maquinaria asegura que, partiendo de $\Gamma_0^{\text{zero}}(\Delta)$ (con Δ maximalmente consistente en el nivel 0), se puede construir una teoría maximalmente consistente Δ_1 en el nivel 1 que extiende los axiomas de Tarski, los axiomas aritméticos y la imagen elevada de Δ, y que además está saturada existencialmente. Esta Δ_1 será el escenario donde se desarrollará el argumento diagonal.

```
--
============================================================
==

-- FASE 1C-α: INFRAESTRUCTURA DE LINDENBAUM Y DIAGONALIZACIÓN PARA s = 0

--
============================================================
==


open Classical


/-- Esquemas axiómaticos de Tarski para el operador de verdad de nivel s+1 -/

def TarskiAxioms (s : Nat) : Set (ArithForm (s + 1)) :=

  { φ' | ∃ (θ : ArithForm s),

    φ' = ArithForm.impl (ArithForm.true_op (s_lower := s) rfl (ArithTerm.const
(NatS_decode (Encodable.encode θ)))) (liftForm θ) } ∪
```

```
 { φ' | ∃ (θ : ArithForm s),
   φ' = ArithForm.impl (liftForm θ) (ArithForm.true_op (s_lower := s) rfl (ArithTerm.const (NatS_decode (Encodable.encode θ)))) }


/-- Conjunto inicial Γ₀ de nivel 1 que incorpora Tarski y Peano -/
def Gamma0_zero (Δ : Set (ArithForm 0)) : Set (ArithForm 1) :=
 { φ' | ∃ φ ∈ Δ, φ' = liftForm φ } ∪
 { φ' | ∃ (x : ArithTerm 1), φ' = add_zero_axiom x } ∪
 { φ' | ∃ (x y : ArithTerm 1), φ' = add_succ_axiom x y } ∪
 TarskiAxioms 0


-- ------------------------------------------------------------------
-- REPLICACIÓN DE LA MAQUINARIA DE HENKIN-LINDENBAUM PARA NIVEL 1
-- ------------------------------------------------------------------


def GammaWitnessesList1 : Nat → List (ArithForm 1)
 | 0 => []
 | m + 1 =>
   match getExistentialComponents (decodeExists (s := 1) m) with
   | some (v, ψ) =>
    let w := Nat.max (max_var_list (GammaWitnessesList1 m)) (max_var_form (decodeExists (s := 1) m)) + 1
    let freshEx := ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)))
    let c := ⟨Nat.max (maxHenkinList (GammaWitnessesList1 m)) (maxHenkinForm freshEx) + 1⟩
    ArithForm.impl freshEx (substForm v (ArithTerm.henkin c) ψ) :: GammaWitnessesList1 m
   | none => GammaWitnessesList1 m
```

```
def GammaSeq1 (Γ : Set (ArithForm 1)) : Nat → Set (ArithForm 1)
 | 0 => Γ
 | m + 1 =>
  match getExistentialComponents (decodeExists (s := 1) m) with
  | some (v, ψ) =>
   let w := Nat.max (max_var_list (GammaWitnessesList1 m)) (max_var_form (decodeExists (s := 1) m)) + 1
   let freshEx := ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)))
   let c := ⟨Nat.max (maxHenkinList (GammaWitnessesList1 m)) (maxHenkinForm freshEx) + 1⟩
   insert (ArithForm.impl freshEx (substForm v (ArithTerm.henkin c) ψ)) (GammaSeq1 Γ m)
  | none => GammaSeq1 Γ m

def GammaLimit1 (Γ : Set (ArithForm 1)) : Set (ArithForm 1) :=
 { φ | ∃ n, φ ∈ GammaSeq1 Γ n }

noncomputable def LindenbaumSeq1 (Γ : Set (ArithForm 1)) : Nat → Set (ArithForm 1)
 | 0 => GammaLimit1 Γ
 | n + 1 =>
  match (Encodable.decode n : Option (ArithForm 1)) with
  | none => LindenbaumSeq1 Γ n
  | some φ =>
   if Consistent (insert φ (LindenbaumSeq1 Γ n)) then
    insert φ (LindenbaumSeq1 Γ n)
   else
    insert (ArithForm.neg φ) (LindenbaumSeq1 Γ n)
```

```
def LindenbaumLimit1 (Γ : Set (ArithForm 1)) : Set (ArithForm 1) :=
  { φ | ∃ n, φ ∈ LindenbaumSeq1 Γ n }
```

Lema `LindenbaumLimit1_maximal`: Maximalidad del límite de Lindenbaum para el nivel 1

Establece que el conjunto límite construido por el procedimiento de Lindenbaum para el nivel 1 es maximal: para toda fórmula φ del nivel 1, o bien φ pertenece al límite, o bien su negación $\neg\varphi$ pertenece al límite:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ \forall \varphi \in \text{ArithForm } 1,\ \varphi \in \text{LindenbaumLimit1}(\Gamma) \lor \neg\varphi \in \text{LindenbaumLimit1}(\Gamma) $$

Intuición y necesidad

La maximalidad es la propiedad definitoria de las extensiones de Lindenbaum: un conjunto es maximal si para cada fórmula, él mismo o su negación está presente, lo que equivale a ser una teoría completa (toda fórmula es decidible dentro de ella). La demostración explota la construcción explícita de la secuencia: dado φ, se considera su código $n = \text{encode}(\varphi)$. Por la propiedad fundamental de la codificación, $\text{decode}(n) = \varphi$. En el paso $n+1$ de la secuencia de Lindenbaum, la definición examina precisamente esta fórmula decodificada. Si φ es consistente con el conjunto acumulado hasta el paso n, se inserta φ; en caso contrario, se inserta $\neg\varphi$. En ambos casos, una de las dos fórmulas pertenece a $\text{LindenbaumSeq1}(\Gamma, n+1)$ y, por tanto, al límite. Esta propiedad garantiza que el límite de Lindenbaum es una teoría completa, requisito indispensable para que pueda servir como la teoría de referencia Δ_1 en el argumento de diagonalización.

Lema `LindenbaumLimit1_contains_Gamma`: El límite de Lindenbaum extiende al conjunto original

Demuestra que el límite de Lindenbaum contiene al conjunto de partida Γ:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ \Gamma \subseteq \text{LindenbaumLimit1}(\Gamma) $$

Intuición y necesidad

Esta propiedad de extensividad garantiza que el proceso de Lindenbaum no pierde fórmulas: todo lo que estaba en el conjunto inicial Γ persiste en el límite final. La demostración es inmediata a partir de las definiciones: cualquier $\varphi \in \Gamma$ pertenece a $\text{GammaSeq1}(\Gamma, 0)$ (por definición de la secuencia en el paso cero), por lo que pertenece a $\text{GammaLimit1}(\Gamma)$, que a su vez coincide con $\text{LindenbaumSeq1}(\Gamma, 0)$, y por tanto pertenece al límite. Esta propiedad es esencial para asegurar que la teoría maximal resultante extiende genuinamente los axiomas de partida (Tarski, Peano, y la elevación de Δ), preservando todo el contenido deductivo original.

Lema `mem_LindenbaumLimit1_of_mem_Δ`: Preservación de las fórmulas de Δ en el nivel superior

Establece que si una fórmula χ pertenece al conjunto maximal Δ del nivel 0, entonces su elevación $\text{liftForm}(\chi)$ pertenece al límite de Lindenbaum del nivel 1 construido a partir de $\Gamma_0^{\text{zero}}(\Delta)$:

$$ \forall \Delta \subseteq \text{ArithForm } 0,\ \forall \chi \in \text{ArithForm } 0,\ \chi \in \Delta \implies \text{liftForm}(\chi) \in \text{LindenbaumLimit1}(\Gamma_0^{\text{zero}}(\Delta)) $$

Intuición y necesidad

Este lema conecta explícitamente la teoría maximal Δ del nivel 0 con la teoría maximal del nivel 1 que se construye a partir de ella. La demostración es composicional: primero se verifica que $\text{liftForm}(\chi)$ pertenece a $\Gamma_0^{\text{zero}}(\Delta)$ por la definición misma de este conjunto (está en la primera componente de la unión). Luego, por el lema de extensividad `LindenbaumLimit1_contains_Gamma`, se concluye que pertenece al límite de Lindenbaum. Este resultado es crucial para la coherencia vertical de la jerarquía estratificada: asegura que todo lo demostrable en el nivel 0 sigue siendo afirmable (vía su elevación) en el nivel 1, condición necesaria para que el predicado de verdad del nivel 1 pueda razonar sobre las fórmulas del nivel 0 de manera fiel.

```
-- ---------------------------------------------------------------------

-- PROPIEDADES DE LINDENBAUM DE NIVEL 1 DEMOSTRADAS RIGUROSAMENTE

-- ---------------------------------------------------------------------


lemma LindenbaumLimit1_maximal (Γ : Set (ArithForm 1)) :

   Maximal (LindenbaumLimit1 Γ) := by

 intro φ

 let n := Encodable.encode φ

 have h_dec : Encodable.decode (α := ArithForm 1) n = some φ := Encodable.encodek
φ

 have h_or : φ ∈ LindenbaumSeq1 Γ (n + 1) ∨ ArithForm.neg φ ∈ LindenbaumSeq1 Γ
(n + 1) := by

   unfold LindenbaumSeq1

   rw [h_dec]

   dsimp
```

```
    split_ifs
    · left; exact Set.mem_insert _ _
    · right; exact Set.mem_insert _ _
  cases h_or with
  | inl h1 => left; exact ⟨n + 1, h1⟩
  | inr h2 => right; exact ⟨n + 1, h2⟩

lemma LindenbaumLimit1_contains_Gamma (Γ : Set (ArithForm 1)) :
    Γ ⊆ LindenbaumLimit1 Γ := by
  intro φ hφ
  use 0
  change φ ∈ GammaLimit1 Γ
  use 0
  exact hφ

lemma mem_LindenbaumLimit1_of_mem_Δ {Δ : Set (ArithForm 0)} (χ : ArithForm 0)
(hχ : χ ∈ Δ) :
      liftForm χ ∈ LindenbaumLimit1 (Gamma0_zero Δ) := by
    have h_in_Gamma0 : liftForm χ ∈ Gamma0_zero Δ := by
      simp only [Gamma0_zero, Set.mem_union, Set.mem_ofPred_eq]
      left; left; left
      exact ⟨χ, hχ, rfl⟩
    apply LindenbaumLimit1_contains_Gamma
    exact h_in_Gamma0
```

Monotonía de las secuencias intermedias del nivel 1

Se establece formalmente que las secuencias constructivas `GammaSeq1` y `LindenbaumSeq1` son monótonas crecientes respecto a la inclusión de conjuntos. Esta propiedad es fundamental para garantizar que el proceso de construcción acumulativa de testigos de Henkin y de extensión de Lindenbaum no pierde fórmulas ya incorporadas en pasos anteriores.

Lema `GammaSeq1_le_succ`: Crecimiento en un paso de la secuencia de Henkin

Establece que cada paso de la secuencia de testigos de Henkin extiende al paso anterior:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ \forall n \in \mathbb{N},\ \text{GammaSeq1}(\Gamma, n) \subseteq \text{GammaSeq1}(\Gamma, n + 1) $$

Intuición y necesidad

La secuencia `GammaSeq1` construye progresivamente una teoría existencialmente saturada añadiendo axiomas de testigo para las fórmulas existenciales encontradas en la enumeración. La monotonía en cada paso es inmediata de la definición: cuando el paso $n+1$ procesa una fórmula con componentes existenciales, el nuevo conjunto se define como la inserción del axioma de testigo en el conjunto anterior; en caso contrario, el conjunto permanece inalterado. En ambos casos, el conjunto del paso n está contenido en el del paso $n+1$. Esta propiedad garantiza que la construcción es acumulativa y que ninguna fórmula se pierde al avanzar en la enumeración.

Lema `GammaSeq1_monotonic`: Monotonía global de la secuencia de Henkin

Generaliza la propiedad anterior a cualquier par de índices ordenados:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ \forall n, m \in \mathbb{N},\ n \leq m \implies \text{GammaSeq1}(\Gamma, n) \subseteq \text{GammaSeq1}(\Gamma, m) $$

Intuición y necesidad

Esta propiedad se obtiene por inducción sobre la derivación de $n \leq m$, aplicando transitivamente el lema `GammaSeq1_le_succ` en cada paso intermedio. Es esencial para razonar sobre el límite `GammaLimit1`: garantiza que la unión de toda la secuencia está bien definida como el supremo de una cadena creciente, y que cualquier fórmula que aparezca en algún paso finito pertenece también a todos los pasos posteriores.

Lema `GammaSeq1_mono_step`: Versión simplificada de la monotonía

Proporciona una formulación directa de la monotonía para su uso en demostraciones:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ \forall a, b \in \mathbb{N},\ a \leq b \implies \text{GammaSeq1}(\Gamma, a) \subseteq \text{GammaSeq1}(\Gamma, b) $$

Intuición y necesidad

Este lema es simplemente una reexposición de `GammaSeq1_monotonic` con nombres de variables ajustados para facilitar su invocación en contextos donde los índices se denotan como a y b. No añade contenido matemático nuevo, pero mejora la ergonomía de la biblioteca de lemas.

Lema `LindenbaumSeq1_subset_succ`: Crecimiento en un paso de la secuencia de Lindenbaum

Establece que cada paso de la extensión de Lindenbaum extiende al paso anterior:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ \forall k \in \mathbb{N},\ \text{LindenbaumSeq1}(\Gamma, k) \subseteq \text{LindenbaumSeq1}(\Gamma, k + 1) $$

Intuición y necesidad

En el paso $k+1$, la secuencia de Lindenbaum examina la k-ésima fórmula en la enumeración de `ArithForm 1`. Si la decodificación falla (caso `none`), el conjunto no se modifica y la inclusión es trivial. Si la decodificación produce una fórmula φ, el algoritmo decide si insertar φ o $\neg\varphi$ basándose en la preservación de la consistencia. En ambos casos, el nuevo conjunto se define como `insert` de alguna fórmula sobre el conjunto anterior, por lo que la inclusión está garantizada. Esta monotonía es crucial para que el límite `LindenbaumLimit1` esté bien definido como unión de una cadena creciente.

Lema `LindenbaumSeq1_monotonic`: Monotonía global de la secuencia de Lindenbaum

Generaliza la propiedad de crecimiento a cualquier par de índices ordenados:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ \forall n, m \in \mathbb{N},\ n \leq m \implies \text{LindenbaumSeq1}(\Gamma, n) \subseteq \text{LindenbaumSeq1}(\Gamma, m) $$

Intuición y necesidad

La demostración procede por inducción sobre la derivación de $n \leq m$, aplicando transitivamente `LindenbaumSeq1_subset_succ` en cada paso. Esta propiedad garantiza que la construcción de Lindenbaum es un proceso de extensión progresiva y

acumulativa: las decisiones tomadas en pasos anteriores (añadir φ o $\neg\varphi$) se preservan en todos los pasos posteriores. Esto es indispensable para demostrar que el límite final es maximalmente consistente, ya que asegura que ninguna fórmula se descarta una vez incorporada, y que el conjunto crece de manera controlada hasta decidir todas las fórmulas del lenguaje.

```
open Classical


--
==========================================================================
==
-- MONOTONÍA DE LAS SECUENCIAS INTERMEDIAS (NIVEL 1)
--
==========================================================================
==


lemma GammaSeq1_le_succ (Γ : Set (ArithForm 1)) (n : Nat) :
   GammaSeq1 Γ n ⊆ GammaSeq1 Γ (n + 1) := by
  intro x hx
  unfold GammaSeq1
  split
  · exact Or.inr hx
  · exact hx


lemma GammaSeq1_monotonic (Γ : Set (ArithForm 1)) (n m : Nat) (h : n ≤ m) :
   GammaSeq1 Γ n ⊆ GammaSeq1 Γ m := by
  induction h with
  | refl => exact Set.Subset.refl _
```

```
  | step _ ih => exact Set.Subset.trans ih (GammaSeq1_le_succ Γ _)

lemma GammaSeq1_mono_step (Γ : Set (ArithForm 1)) {a b : Nat} (h : a ≤ b) :
    GammaSeq1 Γ a ⊆ GammaSeq1 Γ b :=
  GammaSeq1_monotonic Γ a b h

lemma LindenbaumSeq1_subset_succ (Γ : Set (ArithForm 1)) (k : Nat) :
    LindenbaumSeq1 Γ k ⊆ LindenbaumSeq1 Γ (k + 1) := by
  change LindenbaumSeq1 Γ k ⊆ match (Encodable.decode k : Option (ArithForm 1)) with
    | none => LindenbaumSeq1 Γ k
    | some φ => if Consistent (insert φ (LindenbaumSeq1 Γ k)) then insert φ (LindenbaumSeq1 Γ k) else insert (ArithForm.neg φ) (LindenbaumSeq1 Γ k)
  split
  · exact Set.Subset.refl _
  · rename_i φ heq
    split_ifs
    · exact Set.subset_insert φ _
    · exact Set.subset_insert (ArithForm.neg φ) _

lemma LindenbaumSeq1_monotonic (Γ : Set (ArithForm 1)) (n m : Nat) (h : n ≤ m) :
    LindenbaumSeq1 Γ n ⊆ LindenbaumSeq1 Γ m := by
  induction h with
  | refl => exact Set.Subset.refl _
  | step _ ih =>
    exact Set.Subset.trans ih (LindenbaumSeq1_subset_succ Γ _)
```

Lemas de soporte finito con unificación correcta de membresía para el nivel 1

Se establecen dos lemas fundamentales que conectan la pertenencia a los conjuntos límite (definidos como uniones infinitas) con la pertenencia a pasos finitos de las secuencias constructivas. Estos lemas son esenciales para reducir razonamientos sobre conjuntos potencialmente infinitos a argumentos finitarios, donde las propiedades de consistencia y deductivas pueden verificarse paso a paso.

Lema `GammaLimit1_subset_GammaSeq1`: Reducción finitaria del límite de Henkin

Establece que cualquier conjunto finito de fórmulas contenido en el límite de Henkin está ya contenido en algún paso finito de la secuencia:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ \forall \Gamma_0 \subseteq_{\text{fin}} \text{ArithForm } 1,\ \Gamma_0 \subseteq \text{GammaLimit1}(\Gamma) \implies \exists N \in \mathbb{N},\ \Gamma_0 \subseteq \text{GammaSeq1}(\Gamma, N) $$

Intuición y necesidad

El límite de Henkin `GammaLimit1` se define como la unión de todos los conjuntos $\text{GammaSeq1}(\Gamma, n)$ para $n \in \mathbb{N}$. Por tanto, cada fórmula individual en el límite pertenece a algún paso finito. Sin embargo, cuando se tiene un conjunto finito Γ_0 de fórmulas, cada una podría pertenecer a un paso distinto. Este lema demuestra que siempre es posible encontrar un índice N suficientemente grande (concretamente, el máximo de los índices individuales) tal que todas las fórmulas de Γ_0 pertenecen simultáneamente a $\text{GammaSeq1}(\Gamma, N)$. La demostración procede por inducción sobre la estructura finita de Γ_0 (representado como `Finset`): para cada fórmula φ añadida al conjunto acumulado, se obtiene su índice n_φ de pertenencia, se toma el máximo con el índice n_s del resto del conjunto, y se utiliza la monotonía de `GammaSeq1` para elevar ambas pertenencias al paso máximo. Esta propiedad de compacidad sintáctica es indispensable para los argumentos de consistencia: si una contradicción se derivara del límite, se derivaría de un conjunto finito de sus fórmulas, y por tanto de algún paso finito de la secuencia.

Lema `LindenbaumLimit1_subset_LindenbaumSeq1`: Reducción finitaria del límite de Lindenbaum

Establece el análogo del lema anterior para la secuencia de Lindenbaum:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ \forall \Gamma_0 \subseteq_{\text{fin}} \text{ArithForm } 1,\ \Gamma_0 \subseteq \text{LindenbaumLimit1}(\Gamma) \implies \exists N \in \mathbb{N},\ \Gamma_0 \subseteq \text{LindenbaumSeq1}(\Gamma, N) $$

Intuición y necesidad

La estructura de la demostración es idéntica a la del lema anterior, utilizando ahora la monotonía de `LindenbaumSeq1`. Dado un conjunto finito Γ_0 contenido en el límite de Lindenbaum, cada fórmula $\varphi \in \Gamma_0$ pertenece a algún $\text{LindenbaumSeq1}(\Gamma, n_\varphi)$. Tomando $N = \max\{n_\varphi : \varphi \in \Gamma_0\}$, la monotonía garantiza que todas pertenecen a $\text{LindenbaumSeq1}(\Gamma, N)$. Este lema es crucial para demostrar que el límite de Lindenbaum preserva la consistencia: si el límite fuera inconsistente, existiría una prueba finita de $\bot$ a partir de un subconjunto finito de sus fórmulas; este lema reduciría esa inconsistencia a algún paso finito de la secuencia, pero cada paso se construye explícitamente preservando la consistencia, generando una contradicción. La reducción finitaria es, por tanto, el puente lógico entre la construcción paso a paso (donde la consistencia se controla explícitamente) y el límite infinito (donde se necesita garantizar la consistencia global).

--
==========
==

-- LEMAS DE SOPORTE FINITO CON UNIFICACIÓN CORRECTA DE MEMBRESÍA (NIVEL 1)

```
--
==========================================================================
==


lemma GammaLimit1_subset_GammaSeq1 (Γ : Set (ArithForm 1)) (Γ0 : Finset
(ArithForm 1)) :

  ((Γ0 : Set (ArithForm 1)) ⊆ GammaLimit1 Γ) → ∃ N, (Γ0 : Set (ArithForm 1)) ⊆
GammaSeq1 Γ N := by

 classical

 induction Γ0 using Finset.induction_on with

 | empty =>

  intro _

  use 0

  rw [Finset.coe_empty]

  exact Set.empty_subset _

 | insert φ s_fin _ ih =>

  intro h_sub

  have h_phi : φ ∈ GammaLimit1 Γ := h_sub (Finset.mem_coe.mpr
(Finset.mem_insert_self φ s_fin))

  have h_s : (s_fin : Set (ArithForm 1)) ⊆ GammaLimit1 Γ := by

   intro x hx

   apply h_sub

   apply Finset.mem_coe.mpr

   exact Finset.mem_insert_of_mem (Finset.mem_coe.mp hx)

  have h_phi_unfolded : ∃ n, φ ∈ GammaSeq1 Γ n := h_phi

  rcases h_phi_unfolded with ⟨n_phi, hn_phi⟩

  rcases ih h_s with ⟨n_s, hn_s⟩

  use Nat.max n_phi n_s

  intro x hx

  have h_mem := Finset.mem_coe.mp hx
```

```
  simp only [Finset.mem_insert] at h_mem
  cases h_mem with
  | inl h_eq =>
    subst h_eq
    apply GammaSeq1_mono_step Γ (Nat.le_max_left n_phi n_s)
    exact hn_phi
  | inr h_in =>
    apply GammaSeq1_mono_step Γ (Nat.le_max_right n_phi n_s)
    exact hn_s (Finset.mem_coe.mpr h_in)

lemma LindenbaumLimit1_subset_LindenbaumSeq1 (Γ : Set (ArithForm 1)) (Γ0 : Finset (ArithForm 1)) :
  ((Γ0 : Set (ArithForm 1)) ⊆ LindenbaumLimit1 Γ) → ∃ N, (Γ0 : Set (ArithForm 1)) ⊆ LindenbaumSeq1 Γ N := by
 classical
 induction Γ0 using Finset.induction_on with
 | empty =>
  intro _
  use 0
  rw [Finset.coe_empty]
  exact Set.empty_subset _
 | insert φ s_fin _ ih =>
  intro h_sub
  have h_phi : φ ∈ LindenbaumLimit1 Γ := h_sub (Finset.mem_coe.mpr (Finset.mem_insert_self φ s_fin))
  have h_s : (s_fin : Set (ArithForm 1)) ⊆ LindenbaumLimit1 Γ := by
    intro x hx
    apply h_sub
    apply Finset.mem_coe.mpr
```

```
    exact Finset.mem_insert_of_mem (Finset.mem_coe.mp hx)
  have h_phi_unfolded : ∃ n, φ ∈ LindenbaumSeq1 Γ n := h_phi
  rcases h_phi_unfolded with ⟨n_phi, hn_phi⟩
  rcases ih h_s with ⟨n_s, hn_s⟩
  use Nat.max n_phi n_s
  intro x hx
  have h_mem := Finset.mem_coe.mp hx
  simp only [Finset.mem_insert] at h_mem
  cases h_mem with
  | inl h_eq =>
    subst h_eq
    apply LindenbaumSeq1_monotonic Γ n_phi (Nat.max n_phi n_s) (Nat.le_max_left _
_)
    exact hn_phi
  | inr h_in =>
    apply LindenbaumSeq1_monotonic Γ n_s (Nat.max n_phi n_s) (Nat.le_max_right _ _)
    exact hn_s (Finset.mem_coe.mpr h_in)
```

Consistencia de los límites de secuencias para el nivel 1

Se demuestra rigurosamente que las construcciones límite de Henkin y Lindenbaum preservan la consistencia, siempre que cada paso finito de la secuencia subyacente sea consistente. Estos teoremas constituyen la verificación final de que el procedimiento de extensión maximal genera efectivamente una teoría consistente y completa.

Lema `GammaLimit1_consistent`: Consistencia del límite de Henkin

Establece que si todos los pasos finitos de la secuencia de Henkin son consistentes, entonces el límite también lo es:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ (\forall n \in \mathbb{N},\ \text{Consistent}(\text{GammaSeq1}(\Gamma, n))) \implies \text{Consistent}(\text{GammaLimit1}(\Gamma)) $$

Intuición y necesidad

La consistencia es una propiedad compacta en lógica de primer orden: una teoría es inconsistente si y solo si algún subconjunto finito suyo lo es. La demostración explota este principio mediante el lema de soporte finito del sistema deductivo (`SPA_Deduction_Internal_finite_support`). Si el límite fuera inconsistente, existiría una derivación de $\bot$ a partir de un conjunto finito $\Gamma_0 \subseteq \text{GammaLimit1}(\Gamma)$. Por el lema de reducción finitaria `GammaLimit1_subset_GammaSeq1`, existe un paso N tal que $\Gamma_0 \subseteq \text{GammaSeq1}(\Gamma, N)$. Por monotonía del sistema deductivo, la misma derivación de $\bot$ puede realizarse a partir de $\text{GammaSeq1}(\Gamma, N)$, contradiciendo la hipótesis de que cada paso es consistente. Este lema garantiza que el proceso de adición de testigos de Henkin no introduce inconsistencias en el límite, siempre que cada adición individual sea segura.

Lema `LindenbaumSeq1_consistent`: Consistencia de cada paso de la secuencia de Lindenbaum

Demuestra por inducción que cada paso de la extensión de Lindenbaum preserva la consistencia:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ (\forall n \in \mathbb{N},\ \text{Consistent}(\text{GammaSeq1}(\Gamma, n))) \implies \forall n \in \mathbb{N},\ \text{Consistent}(\text{LindenbaumSeq1}(\Gamma, n)) $$

Intuición y necesidad

La demostración procede por inducción sobre el índice n:

- **Caso base ($n = 0$):** $\text{LindenbaumSeq1}(\Gamma, 0) = \text{GammaLimit1}(\Gamma)$, que es consistente por el lema anterior.

- **Paso inductivo ($k \to k+1$):** En el paso $k+1$, se examina la k-ésima fórmula φ de la enumeración. Se presentan dos casos:

  - Si la decodificación falla, el conjunto no se modifica y la consistencia se hereda de la hipótesis de inducción.

  - Si la decodificación produce φ, el algoritmo evalúa si φ es consistente con el conjunto acumulado. Si lo es, se inserta φ (y la consistencia está garantizada por la condición del condicional). Si no lo es, se inserta $\neg\varphi$. En este último caso, se invoca el lema `consistent_insert_neg`, que establece precisamente que si una fórmula no es consistente con un conjunto, entonces su negación sí lo es, preservando la consistencia global.

Este lema es el corazón de la construcción de Lindenbaum: verifica que la decisión dicotómica de añadir φ o $\neg\varphi$ en cada paso nunca compromete la consistencia, independientemente del orden de enumeración de las fórmulas.

Lema `LindenbaumLimit1_consistent`: Consistencia del límite de Lindenbaum

Establece el resultado principal: el límite final del procedimiento de Lindenbaum es consistente:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ (\forall n \in \mathbb{N},\ \text{Consistent}(\text{GammaSeq1}(\Gamma, n))) \implies \text{Consistent}(\text{LindenbaumLimit1}(\Gamma)) $$

Intuición y necesidad

La demostración replica la estructura del lema `GammaLimit1_consistent`, utilizando ahora la reducción finitaria de Lindenbaum. Si el límite fuera inconsistente, existiría un subconjunto finito $\Gamma_0 \subseteq \text{LindenbaumLimit1}(\Gamma)$ que prueba $\bot$. Por el lema `LindenbaumLimit1_subset_LindenbaumSeq1`, existe un paso N tal que $\Gamma_0 \subseteq \text{LindenbaumSeq1}(\Gamma, N)$. Por monotonía deductiva, $\bot$ sería demostrable a partir de $\text{LindenbaumSeq1}(\Gamma, N)$, pero esto contradice el lema anterior que garantiza la consistencia de cada paso individual. Este teorema corona la construcción de Lindenbaum: partiendo de una teoría consistente Γ, el procedimiento genera una extensión $\text{LindenbaumLimit1}(\Gamma)$ que es simultáneamente consistente (por este lema) y maximal (por `LindenbaumLimit1_maximal`), es decir, una teoría completa. Esta teoría completa será el escenario Δ_1 donde se desarrollará el argumento de diagonalización para demostrar la no representabilidad de ciertas funciones.

```
--
============================================================
==
-- CONSISTENCIA DE LOS LÍMITES DE SECUENCIAS (NIVEL 1)
--
============================================================
===


lemma GammaLimit1_consistent (Γ : Set (ArithForm 1))
    (h_con : ∀ n, Consistent (GammaSeq1 Γ n)) : Consistent (GammaLimit1 Γ) := by
  intro h_bot
  obtain ⟨Γ0, h_sub, h_ded⟩ := SPA_Deduction_Internal_finite_support h_bot
  obtain ⟨N, h_seq_sub⟩ := GammaLimit1_subset_GammaSeq1 Γ Γ0 h_sub
  have h_ded_N : SPA_Deduction_Internal 1 (GammaSeq1 Γ N) ArithForm.bottom := by
    apply SPA_Deduction_Internal.weaken (Γ0 : Set (ArithForm 1))
    · exact h_ded
```

```
  · exact h_seq_sub
 exact h_con N h_ded_N

lemma LindenbaumSeq1_consistent (Γ : Set (ArithForm 1))
   (h_con_seq : ∀ n, Consistent (GammaSeq1 Γ n)) :
   ∀ n, Consistent (LindenbaumSeq1 Γ n) := by
 intro n
 induction n with
 | zero =>
  exact GammaLimit1_consistent Γ h_con_seq
 | succ k ih =>
  change Consistent (match (Encodable.decode k : Option (ArithForm 1)) with
    | none => LindenbaumSeq1 Γ k
   | some φ => if Consistent (insert φ (LindenbaumSeq1 Γ k)) then insert φ
(LindenbaumSeq1 Γ k) else insert (ArithForm.neg φ) (LindenbaumSeq1 Γ k))
  split
  · exact ih
  · rename_i φ heq
    split_ifs with h_cond
    · exact h_cond
    · exact consistent_insert_neg (LindenbaumSeq1 Γ k) ih φ h_cond

lemma LindenbaumLimit1_consistent (Γ : Set (ArithForm 1))
   (h_con_seq : ∀ n, Consistent (GammaSeq1 Γ n)) : Consistent (LindenbaumLimit1 Γ)
:= by
 intro h_bot
 obtain ⟨Γ0, h_sub, h_ded⟩ := SPA_Deduction_Internal_finite_support h_bot
 obtain ⟨N, h_seq_sub⟩ := LindenbaumLimit1_subset_LindenbaumSeq1 Γ Γ0 h_sub
```

```
  have h_ded_N : SPA_Deduction_Internal 1 (LindenbaumSeq1 Γ N) ArithForm.bottom
:= by
    apply SPA_Deduction_Internal.weaken (Γ0 : Set (ArithForm 1))
    · exact h_ded
    · exact h_seq_sub
  exact LindenbaumSeq1_consistent Γ h_con_seq N h_ded_N
```

Lema `LindenbaumLimit1_Henkin`: Propiedad de Henkin del límite de Lindenbaum

Establece que el límite del procedimiento de Lindenbaum para el nivel 1 satisface la propiedad de Henkin: para toda fórmula existencial demostrable en el límite, existe un testigo cerrado (una constante de Henkin) que la satisface dentro del mismo límite:

$$ \forall \Gamma \subseteq \text{ArithForm } 1,\ (\forall n \in \mathbb{N},\ \text{Consistent}(\text{GammaSeq1}(\Gamma, n))) \implies \text{HenkinSet}(\text{LindenbaumLimit1}(\Gamma)) $$

Formalmente, la propiedad de Henkin establece:

$$ \forall v \in \mathbb{N},\ \forall \psi \in \text{ArithForm } 1,\ \neg \forall v \neg \psi \in \text{LindenbaumLimit1}(\Gamma) \implies \exists c \in \text{HenkinConst},\ \psi[v := \text{henkin}(c)] \in \text{LindenbaumLimit1}(\Gamma) $$

Intuición y necesidad

La propiedad de Henkin es la contraparte sintáctica de la saturación existencial semántica: garantiza que si una fórmula existencial $\exists v \psi$ (expresada como $\neg \forall v \neg \psi$) es afirmada por la teoría, entonces existe un testigo concreto —una constante de Henkin c— tal que la instancia $\psi[v := c]$ también es afirmada. Esta propiedad es indispensable para la construcción del modelo canónico en la prueba de completitud, pues asegura que los testigos necesarios para validar todas las afirmaciones existenciales están disponibles sintácticamente dentro de la teoría.

La demostración distingue dos casos según si la variable v aparece ligada en ψ o no.

Caso estándar: v no ocurre ligada en ψ

Cuando $\text{NoBoundVar}(v, \psi)$, la variable existencial no presenta conflictos de ligadura, y la construcción del testigo procede directamente mediante la maquinaria de Henkin previamente establecida:

1. **Construcción de la fórmula existencial canónica:** Se define $\varphi := \neg \forall v \neg \psi$, que es la representación de $\exists v \psi$ en el lenguaje.

2. **Localización en la enumeración:** Sea $code = \text{encode}(\varphi)$. Por definición de la secuencia de Henkin, en el paso $code + 1$ se procesa precisamente esta fórmula, y sus componentes existenciales se identifican como (v, ψ).

3. **Generación de símbolos frescos:** Se construye una variable fresca w y una constante de Henkin fresca c cuyos índices superan todas las variables y constantes utilizadas hasta el momento. Las propiedades $v \neq w$ y $\text{NotOccurForm}(w, \psi)$ se demuestran utilizando la cota superior provista por los operadores `max_var_form` y `max_var_list`.

4. **Incorporación del axioma de testigo:** El axioma $\text{freshEx} \to \psi[v := \text{henkin}(c)]$ (donde $\text{freshEx} = \neg \forall w \neg \psi[v := \text{var }

w]$) pertenece a $\text{GammaSeq1}(\Gamma, code + 1)$ por construcción, y por tanto al límite de Lindenbaum.

5. **Equivalencia por cambio de variable:** El lema `change_var_equivalence_strong` establece que, en una teoría maximal consistente, la fórmula $\neg \forall v \neg \psi$ es equivalente a $\neg \forall w \neg \psi[v := \text{var } w]$. Aplicando esta equivalencia a la hipótesis h_exists, se obtiene que freshEx pertenece al límite.

6. **Modus ponens maximal:** Finalmente, puesto que tanto freshEx como el axioma $\text{freshEx} \to \psi[v := \text{henkin}(c)]$ pertenecen al límite, y el límite es maximal consistente, se concluye que $\psi[v := \text{henkin}(c)]$ también pertenece al límite. El testigo $\text{henkin}(c)$ es cerrado por construcción.

Caso patológico: v está ligada en ψ

Cuando v aparece ligada en ψ, la construcción directa no es posible porque la sustitución ingenua podría causar captura de variables. La estrategia consiste en renombrar la variable ligada problemática antes de aplicar el caso estándar:

1. **Renombramiento alpha:** Se elige una variable fresca u (típicamente $u = \text{max_var_form}(\psi) + v + 1$) que no ocurre en ψ y es distinta de v. Se considera $\psi' := \text{renameBoundVar}(v, u, \psi)$, que es α-equivalente a ψ pero tiene u en lugar de v como variable ligada.

2. **Preservación de la existencia:** El lema de α-equivalencia (`alpha_rename_inside`) garantiza que $\neg \forall v \neg \psi \to \neg \forall v \neg \psi'$ es demostrable. Como el límite es maximal consistente y contiene $\neg \forall v \neg \psi$ (por hipótesis), también contiene $\neg \forall v \neg \psi'$.

3. **Aplicación del caso estándar a ψ':** Dado que por construcción v no ocurre ligada en ψ' (pues la variable ligada renombrada es u, no v), se aplica

íntegramente el caso estándar a ψ', obteniendo una constante de Henkin c' tal que $\psi'[v := \text{henkin}(c')] \in \text{LindenbaumLimit1}(\Gamma)$.

4. **Transferencia del testigo de vuelta a ψ:** El lema `rename_equiv` establece la equivalencia entre $\psi'[v := t]$ y $\psi[v := t]$ para cualquier término cerrado t, siempre que u no ocurra en t (lo cual se cumple para constantes de Henkin frescas). En particular, se obtiene que $\psi'[v := \text{henkin}(c')] \to \psi[v := \text{henkin}(c')]$ es demostrable. Como el límite es maximal consistente, se concluye que $\psi[v := \text{henkin}(c')]$ pertenece al límite.

Este lema completa la verificación de que el límite de Lindenbaum no solo es maximal y consistente, sino también existencialmente saturado, cumpliendo todos los requisitos para ser una teoría de Henkin sobre la cual se puede construir un modelo canónico en la prueba de completitud.

```
lemma LindenbaumLimit1_Henkin (Γ : Set (ArithForm 1)) (h_con_seq : ∀ n, Consistent (GammaSeq1 Γ n)) :

  HenkinSet (LindenbaumLimit1 Γ) := by

 intro v ψ h_exists

 have h_max : Maximal (LindenbaumLimit1 Γ) := LindenbaumLimit1_maximal Γ

 have h_con : Consistent (LindenbaumLimit1 Γ) := LindenbaumLimit1_consistent Γ h_con_seq

 by_cases h_bound : NoBoundVar v ψ

 · -- CASO ESTÁNDAR: v no ocurre ligada en ψ

  let φ := ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))

  let code := Encodable.encode φ

  have h_code : decodeExists (s := 1) code = φ := by

   unfold decodeExists

   have h_encode : Encodable.decode (α := ArithForm 1) code = some φ := Encodable.encodek φ

   rw [h_encode]
```

```
    rfl
  have h_comp : getExistentialComponents (decodeExists (s := 1) code) = some (v, ψ)
:= by
    rw [h_code]
    rfl
  let w := Nat.max (max_var_list (GammaWitnessesList1 code)) (max_var_form
(decodeExists (s := 1) code)) + 1
  have h_vw : v ≠ w := by
    have h_max_form : max_var_form (decodeExists (s := 1) code) ≥ v := by
      rw [h_code]
      change Nat.max v (max_var_form ψ) ≥ v
      exact Nat.le_max_left v (max_var_form ψ)
    have hw_gt : w > max_var_form (decodeExists (s := 1) code) := by
      dsimp [w]
      have : max_var_form (decodeExists (s := 1) code) ≤ Nat.max (max_var_list
(GammaWitnessesList1 code)) (max_var_form (decodeExists (s := 1) code)) :=
Nat.le_max_right _ _
      omega
    omega
  have h_w_ψ : NotOccurForm w ψ := by
    have h_max_form : max_var_form (decodeExists (s := 1) code) ≥ max_var_form ψ :=
by
      rw [h_code]
      change Nat.max v (max_var_form ψ) ≥ max_var_form ψ
      exact Nat.le_max_right v (max_var_form ψ)
    have hw_gt : w > max_var_form (decodeExists (s := 1) code) := by
      dsimp [w]
      have : max_var_form (decodeExists (s := 1) code) ≤ Nat.max (max_var_list
(GammaWitnessesList1 code)) (max_var_form (decodeExists (s := 1) code)) :=
Nat.le_max_right _ _
      omega
```

exact NotOccurForm_of_gt_max_var ψ w (by omega)

let freshEx := ArithForm.neg (ArithForm.forall_form w (ArithForm.neg (substForm v (ArithTerm.var w) ψ)))

let c : HenkinConst := ⟨Nat.max (maxHenkinList (GammaWitnessesList1 code)) (maxHenkinForm freshEx) + 1⟩

let ax := ArithForm.impl freshEx (substForm v (ArithTerm.henkin c) ψ)

have h_ax_Linden : ax ∈ LindenbaumLimit1 Γ := by

have h_ax_GammaSeq : ax ∈ GammaSeq1 Γ (code + 1) := by

unfold GammaSeq1

rw [h_comp]

exact Or.inl rfl

have h_ax_GammaLimit : ax ∈ GammaLimit1 Γ := ⟨code + 1, h_ax_GammaSeq⟩

exact ⟨0, h_ax_GammaLimit⟩

have h_equiv := (change_var_equivalence_strong (LindenbaumLimit1 Γ) h_max h_con v w ψ h_vw h_w_ψ h_bound).mp h_exists

have h_target : substForm v (ArithTerm.henkin c) ψ ∈ LindenbaumLimit1 Γ :=

maximal_consistent_imp_closed (LindenbaumLimit1 Γ) h_max h_con freshEx (substForm v (ArithTerm.henkin c) ψ) h_equiv h_ax_Linden

exact ⟨henkin_closed c, h_target⟩

· -- CASO PATOLÓGICO: v está ligada en ψ

let u := max_var_form ψ + v + 1

have h_vu : v ≠ u := by omega

have h_u_ψ : NotOccurForm u ψ := NotOccurForm_of_gt_max_var ψ u (by omega)

have h_bound' : NoBoundVar v (renameBoundVar v u ψ) := NoBoundVar_renameBoundVar ψ v u h_vu

have h_impl := alpha_rename_inside v u ψ h_vu h_u_ψ

have h_impl_Linden_in : ArithForm.impl (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ)))) ∈ LindenbaumLimit1 Γ := by

have h_impl_Linden : SPA_Deduction_Internal 1 (LindenbaumLimit1 Γ) (ArithForm.impl (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ)))

(ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ))))) := by

apply SPA_Deduction_Internal.weaken ∅ _ (LindenbaumLimit1 Γ) h_impl (Set.empty_subset _)

exact maximal_consistent_closed_under_deduction (LindenbaumLimit1 Γ) h_max h_con _ h_impl_Linden

have h_exists' : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ))) ∈ LindenbaumLimit1 Γ :=

maximal_consistent_imp_closed (LindenbaumLimit1 Γ) h_max h_con (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ))) (ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (renameBoundVar v u ψ)))) h_exists h_impl_Linden_in

-- Aplicamos el caso estándar a ψ' (ya que tiene h_bound' de forma segura)

let ψ' := renameBoundVar v u ψ

let φ' := ArithForm.neg (ArithForm.forall_form v (ArithForm.neg ψ'))

let code' := Encodable.encode φ'

have h_code' : decodeExists (s := 1) code' = φ' := by

unfold decodeExists

have h_encode' : Encodable.decode (α := ArithForm 1) code' = some φ' := Encodable.encodek φ'

rw [h_encode']

rfl

have h_comp' : getExistentialComponents (decodeExists (s := 1) code') = some (v, ψ') := by

rw [h_code']

rfl

let w' := Nat.max (max_var_list (GammaWitnessesList1 code')) (max_var_form (decodeExists (s := 1) code')) + 1

have h_vw' : v ≠ w' := by

have h_max_form' : max_var_form (decodeExists (s := 1) code') ≥ v := by

rw [h_code']

```
      change Nat.max v (max_var_form ψ') ≥ v
      exact Nat.le_max_left v (max_var_form ψ')
    have hw_gt' : w' > max_var_form (decodeExists (s := 1) code') := by
      dsimp [w']
      have : max_var_form (decodeExists (s := 1) code') ≤ Nat.max (max_var_list (GammaWitnessesList1 code')) (max_var_form (decodeExists (s := 1) code')) := Nat.le_max_right _ _
      omega
    omega
  have h_w_ψ' : NotOccurForm w' ψ' := by
    have h_max_form' : max_var_form (decodeExists (s := 1) code') ≥ max_var_form ψ' := by
      rw [h_code']
      change Nat.max v (max_var_form ψ') ≥ max_var_form ψ'
      exact Nat.le_max_right v (max_var_form ψ')
    have hw_gt' : w' > max_var_form (decodeExists (s := 1) code') := by
      dsimp [w']
      have : max_var_form (decodeExists (s := 1) code') ≤ Nat.max (max_var_list (GammaWitnessesList1 code')) (max_var_form (decodeExists (s := 1) code')) := Nat.le_max_right _ _
      omega
    exact NotOccurForm_of_gt_max_var ψ' w' (by omega)
  let freshEx' := ArithForm.neg (ArithForm.forall_form w' (ArithForm.neg (substForm v (ArithTerm.var w') ψ')))
  let c' : HenkinConst := ⟨Nat.max (maxHenkinList (GammaWitnessesList1 code')) (maxHenkinForm freshEx') + 1⟩
  let ax' := ArithForm.impl freshEx' (substForm v (ArithTerm.henkin c') ψ')
  have h_ax_Linden' : ax' ∈ LindenbaumLimit1 Γ := by
    have h_ax_GammaSeq' : ax' ∈ GammaSeq1 Γ (code' + 1) := by
      unfold GammaSeq1
      rw [h_comp']
```

```
      dsimp only
      exact Or.inl rfl
    have h_ax_GammaLimit' : ax' ∈ GammaLimit1 Γ := ⟨code' + 1, h_ax_GammaSeq'⟩
    exact ⟨0, h_ax_GammaLimit'⟩
  have h_equiv' := (change_var_equivalence_strong (LindenbaumLimit1 Γ) h_max h_con v w' ψ' h_vw' h_w_ψ' h_bound').mp h_exists'
  have h_target' : substForm v (ArithTerm.henkin c') ψ' ∈ LindenbaumLimit1 Γ :=
    maximal_consistent_imp_closed (LindenbaumLimit1 Γ) h_max h_con freshEx' (substForm v (ArithTerm.henkin c') ψ') h_equiv' h_ax_Linden'

  -- Usamos rename_equiv para transferir el testigo c' de ψ' de vuelta a ψ
  have h_equiv_target := rename_equiv v u (ArithTerm.henkin c') ψ h_vu h_u_ψ
  have h_equiv_target_Linden_in : (substForm v (ArithTerm.henkin c') (renameBoundVar v u ψ)).impl (substForm v (ArithTerm.henkin c') ψ) ∈ LindenbaumLimit1 Γ := by
    have h_equiv_target_Linden : SPA_Deduction_Internal 1 (LindenbaumLimit1 Γ) (ArithForm.impl (substForm v (ArithTerm.henkin c') (renameBoundVar v u ψ)) (substForm v (ArithTerm.henkin c') ψ)) := by
      apply SPA_Deduction_Internal.weaken ∅ _ (LindenbaumLimit1 Γ) h_equiv_target.left (Set.empty_subset _)
    exact maximal_consistent_closed_under_deduction (LindenbaumLimit1 Γ) h_max h_con _ h_equiv_target_Linden
  have h_target : substForm v (ArithTerm.henkin c') ψ ∈ LindenbaumLimit1 Γ :=
    maximal_consistent_imp_closed (LindenbaumLimit1 Γ) h_max h_con (substForm v (ArithTerm.henkin c') (renameBoundVar v u ψ)) (substForm v (ArithTerm.henkin c') ψ) h_target' h_equiv_target_Linden_in
  exact ⟨henkin_closed c', h_target⟩
```

Lema `liftTerm_substAllTerm`: Conmutación de la elevación con la sustitución total de términos

Establece que la operación de elevación de estrato conmuta con la sustitución simultánea de todas las variables libres en un término. Formalmente, para todo término t en el estrato s y toda asignación de variables `env` $: \mathbb{N} \to \text{ArithTerm } s$:

$$ \forall s \in \mathbb{N},\ \forall t \in \text{ArithTerm } s,\ \forall\ \text{env} : \mathbb{N} \to \text{ArithTerm } s,\ \text{liftTerm}(\text{substAllTerm}(t, \text{env})) = \text{substAllTerm}(\text{liftTerm}(t), \lambda v.\ \text{liftTerm}(\text{env}(v))) $$

Intuición y necesidad

La sustitución total `substAllTerm` reemplaza simultáneamente cada variable libre v en un término por el término $\text{env}(v)$. Este lema generaliza el resultado previo `liftTerm_substTerm` (que trataba la sustitución de una única variable) al caso de sustitución múltiple. La demostración procede por inducción estructural sobre t:

- Los casos base (`var`, `const`, `henkin`, `zero`) son inmediatos pues la sustitución total y la elevación actúan de manera independiente y conmutan trivialmente.

- Los casos recursivos (`succ`, `add`, `mul`, `exp`, `pred`) despliegan las definiciones de `substAllTerm` y `liftTerm`, y aplican las hipótesis de inducción a cada subcomponente, recomponiendo la igualdad.

Este lema es fundamental para la coherencia de la Gödelización: asegura que la elevación de un término donde se han instanciado todas las variables libres es equivalente a elevar primero el término y luego instanciar las variables con las versiones elevadas de los términos de sustitución. Esta propiedad functorial es indispensable para manipular fórmulas con sustitución total en contextos estratificados.

Lema `liftForm_substAll_raw`: Conmutación de la elevación con la sustitución total de fórmulas en el nivel base

Extiende la propiedad de conmutación al nivel de las fórmulas para el caso particular del estrato $s = 0$, donde la sustitución total se define mediante `substAll_raw`:

$$ \forall\ \text{env} : \mathbb{N} \to \text{ArithTerm } 0,\ \forall \varphi \in \text{ArithForm } 0,\ \text{liftForm}(\text{substAll_raw}(\varphi, \text{env})) = \text{substAll_raw}(\text{liftForm}(\varphi), \lambda v.\ \text{liftTerm}(\text{env}(v))) $$

Intuición y necesidad

`substAll_raw` es la operación de sustitución total para fórmulas que reemplaza cada variable libre v por el término $\text{env}(v)$, con la particularidad de que en el caso del cuantificador universal, la variable ligada n se preserva (no se sustituye) y la asignación se actualiza para reflejar que n debe mapearse a sí misma dentro del alcance del cuantificador. La demostración procede por inducción sobre φ con generalización de `env`:

- **Casos atómicos** (`bottom`, `eq`, `list_nth_eq`): Se reducen al lema `liftTerm_substAllTerm` aplicado a los subtérminos, ya que no involucran ligadura de variables.

- **Conectivos proposicionales** (`impl`, `neg`): Aplican directamente las hipótesis de inducción, pues la sustitución total se distribuye sobre los conectivos sin modificar el entorno.

- **Cuantificador universal** (`forall_form n f`): Este es el caso más delicado. La sustitución total `substAll_raw` modifica el entorno dentro del alcance del cuantificador, mapeando n a $\text{var } n$ (pues n está ligada y no debe ser sustituida). Se requiere demostrar que la elevación de este entorno modificado coincide con el entorno modificado después de la elevación, es decir:

$$ \lambda v.\ \text{liftTerm}(\text{if } v = n \text{ then var } n \text{ else env}(v)) = \lambda v.\ \text{if } v = n \text{ then var } n \text{ else liftTerm}(\text{env}(v)) $$

Esta igualdad se verifica por extensionalidad: para cada v, si $v = n$, ambos lados se reducen a $\text{var } n$ (pues `liftTerm` preserva las variables); si $v \neq n$, ambos lados se reducen a $\text{liftTerm}(\text{env}(v))$.

- **Operadores estratificados** (`provable_op`, `true_op`): Se reducen nuevamente a `liftTerm_substAllTerm`, ya que solo contienen un término como subcomponente sintáctico.

Este lema es crucial para la Fase 1C-α porque permite trasladar relaciones de sustitución total entre el nivel 0 y el nivel 1. En particular, cuando se trabaja con la diagonalización y la representabilidad de funciones, es necesario manipular fórmulas donde todas las variables libres han sido instanciadas con numerales o términos cerrados, y la capacidad de elevar estas construcciones preservando la estructura de sustitución es indispensable para la coherencia lógica entre estratos.

```
lemma liftTerm_substAllTerm {s : Nat} (t : ArithTerm s) (env : Nat → ArithTerm s) :
  liftTerm (substAllTerm t env) = substAllTerm (liftTerm t) (fun v => liftTerm (env v))
:= by
 induction t with
 | var n => rfl
 | const c => rfl
 | henkin c => rfl
 | zero => rfl
 | succ t' ih =>
  dsimp [substAllTerm, liftTerm]
  rw [ih]
 | add t1 t2 ih1 ih2 =>
  dsimp [substAllTerm, liftTerm]
  rw [ih1, ih2]
```

```
  | mul t1 t2 ih1 ih2 =>
    dsimp [substAllTerm, liftTerm]
    rw [ih1, ih2]
  | exp t1 t2 ih1 ih2 =>
    dsimp [substAllTerm, liftTerm]
    rw [ih1, ih2]
  | pred t' ih =>
    dsimp [substAllTerm, liftTerm]
    rw [ih]

lemma liftForm_substAll_raw {env : Nat → ArithTerm 0} (φ : ArithForm 0) :
    liftForm (substAll_raw φ env) = substAll_raw (liftForm φ) (fun v => liftTerm (env
v)) := by
  induction φ generalizing env with
  | bottom => rfl
  | eq t1 t2 =>
    dsimp [substAll_raw, liftForm]
    rw [liftTerm_substAllTerm t1, liftTerm_substAllTerm t2]
  | impl f1 f2 ih1 ih2 =>
    dsimp [substAll_raw, liftForm]
    rw [ih1, ih2]
  | neg f ih =>
    dsimp [substAll_raw, liftForm]
    rw [ih]
  | forall_form n f ih =>
    dsimp [substAll_raw, liftForm]
    rw [ih]
    have h_env : (fun v => liftTerm (if v = n then ArithTerm.var n else env v)) =
```

```
          (fun v => if v = n then ArithTerm.var n else liftTerm (env v)) := by
    ext v
    split_ifs with h_vn
    · rfl
    · rfl
  rw [h_env]
| provable_op h t =>
  dsimp [substAll_raw, liftForm]
  rw [liftTerm_substAllTerm]
| true_op h t =>
  dsimp [substAll_raw, liftForm]
  rw [liftTerm_substAllTerm]
| list_nth_eq t1 t2 t3 =>
  dsimp [substAll_raw, liftForm]
  rw [liftTerm_substAllTerm t1, liftTerm_substAllTerm t2, liftTerm_substAllTerm t3]
```

Función `toSPACode`: Conversión estructural entre tipos `Code`

Se define la función de conversión que traduce un código del tipo global `_root_.Code s` (definido en el ámbito superior) al tipo local `SPA.Code s` (definido dentro del espacio de nombres `SPA`). Ambos tipos son estructuralmente idénticos —listas inductivas de números naturales indexadas por un parámetro fantasma de estrato—, pero Lean los distingue por pertenecer a ámbitos de nombres diferentes:

$$ \text{toSPACode} : __root__.\text{Code } s \to \text{SPA.Code } s $$

$$ \text{toSPACode}(\text{nil}) = \text{nil} $$

$$ \text{toSPACode}(\text{cons } x\ xs) = \text{cons } x\ (\text{toSPACode } xs) $$

Intuición y necesidad

Durante el desarrollo modular del sistema SPA, pueden coexistir definiciones de `Code` en diferentes espacios de nombres (por ejemplo, un `Code` genérico en el preámbulo del archivo y un `Code` específico dentro del namespace `SPA`). La función `toSPACode` actúa como puente sintáctico que permite reutilizar las funciones de serialización (`serializeTerm`, `serializeForm`) —que devuelven `_root_.Code`— dentro del contexto de `SPA`, donde las definiciones subsecuentes (`internalCode`, `encodeTerm`, `encodeForm`) esperan `SPA.Code`. La conversión es puramente estructural y no altera la información subyacente.

Lema `toSPACode_injective`: Inyectividad de la conversión

Demuestra que la función de conversión `toSPACode` es inyectiva: códigos distintos del tipo global se mapean a códigos distintos del tipo local:

$$ \forall s \in \mathbb{N},\ \forall c_1, c_2 \in __root__.\text{Code } s,\ \text{toSPACode}(c_1) = \text{toSPACode}(c_2) \implies c_1 = c_2 $$

Intuición y necesidad

La inyectividad de `toSPACode` es indispensable para garantizar que la cadena de codificaciones que culmina en `encodeTerm` y `encodeForm` preserva la información sin colapsos. La demostración procede por inducción sobre c_1 con generalización de c_2:

- Si $c_1 = \text{nil}$, entonces c_2 debe ser también nil, pues en caso contrario ($c_2 = \text{cons } y\ ys$) se tendría $\text{nil} = \text{cons } y\ (\text{toSPACode } ys)$, lo cual es imposible por la estructura inductiva de `Code`.

- Si $c_1 = \text{cons } x\ xs$, entonces c_2 debe ser de la forma $\text{cons } y\ ys$. Desplegando la definición de `toSPACode` e inyectando la igualdad, se obtiene $x = y$ y $\text{toSPACode}(xs) = \text{toSPACode}(ys)$. La hipótesis de inducción aplicada a xs e ys permite concluir $xs = ys$, y por tanto $c_1 = c_2$.

Funciones `encodeTerm` y `encodeForm`: Codificación completa de términos y fórmulas

Se definen las funciones de codificación que integran la serialización, la conversión de tipos y la codificación interna para producir un término cerrado del nivel s a partir de un término o fórmula del nivel $s+1$:

$$ \text{encodeTerm} : \text{ArithTerm}(s+1) \to \text{TermClosed } s $$

$$ \text{encodeTerm}(t) := \text{internalCode}(\text{toSPACode}(\text{serializeTerm}(t))) $$

$$ \text{encodeForm} : \text{ArithForm}(s+1) \to \text{TermClosed } s $$

$$ \text{encodeForm}(\varphi) := \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\varphi))) $$

Intuición y necesidad

Estas funciones constituyen la realización concreta de la Gödelización en el sistema estratificado SPA. El proceso de codificación sigue tres etapas:

1. **Serialización** (`serializeTerm` / `serializeForm`): Convierte el objeto sintáctico en una lista plana de números naturales (tipo `_root_.Code s`), preservando toda la información estructural mediante un esquema de codificación determinista.

2. **Conversión de tipo** (`toSPACode`): Traduce el código del tipo global al tipo local `SPA.Code s`, necesario para la interoperabilidad dentro del namespace `SPA`.

3. **Codificación interna** (`internalCode`): Convierte el código (ya de tipo `SPA.Code (s+1)`) en un numeral de Peano cerrado del nivel s, utilizando la instancia `Encodable` para obtener el número natural correspondiente y `natToTerm` para representarlo como término sintáctico.

El desplazamiento de niveles es deliberado: un término o fórmula del nivel $s+1$ se codifica como un término cerrado del nivel s. Esta inversión refleja la jerarquía estratificada donde los códigos de objetos de un nivel residen en el nivel inmediatamente inferior, permitiendo que el nivel s hable sobre la sintaxis del nivel $s+1$ sin violar la disciplina de estratificación.

Lemas de inyectividad para la serialización

Se demuestra que las funciones de serialización son inyectivas, aprovechando las propiedades de parseo establecidas en fases anteriores:

$$ \forall s \in \mathbb{N},\ \forall t_1, t_2 \in \text{ArithTerm } s,\ \text{serializeTerm}(t_1) = \text{serializeTerm}(t_2) \implies t_1 = t_2 $$

$$ \forall s \in \mathbb{N},\ \forall \varphi_1, \varphi_2 \in \text{ArithForm } s,\ \text{serializeForm}(\varphi_1) = \text{serializeForm}(\varphi_2) \implies \varphi_1 = \varphi_2 $$

Intuición y necesidad

La inyectividad de la serialización es la propiedad fundamental que garantiza que la codificación es una asignación unívoca: objetos sintácticos distintos reciben códigos distintos. La demostración es elegante y explota la existencia de los decodificadores: si $\text{serializeTerm}(t_1) = \text{serializeTerm}(t_2)$, aplicando `decodeTerm` a ambos lados y usando el lema de corrección `decodeTerm_encodeTerm` (que establece $\text{decodeTerm}(\text{serializeTerm}(t)) = \text{some } t$), se obtiene $\text{some } t_1 = \text{some } t_2$, de donde $t_1 = t_2$ por inyección. El mismo argumento se aplica mutatis mutandis para fórmulas usando `decodeForm_encodeForm`. Esta propiedad es la piedra angular de la Gödelización: sin inyectividad, la codificación no sería una representación fiel de la sintaxis.

Lemas de inyectividad para la codificación completa

Se demuestra que las funciones `encodeTerm` y `encodeForm` son inyectivas, combinando las inyectividades de cada etapa del proceso:

$$ \forall s \in \mathbb{N},\ \text{Function.Injective } (\text{encodeTerm}_{(s:=s)}) $$

$$ \forall s \in \mathbb{N},\ \text{Function.Injective } (\text{encodeForm}_{(s:=s)}) $$

Intuición y necesidad

La demostración descompone la igualdad $\text{encodeTerm}(t_1) = \text{encodeTerm}(t_2)$ en sus tres componentes:

1. Por definición, $\text{internalCode}(\text{toSPACode}(\text{serializeTerm}(t_1))) = \text{internalCode}(\text{toSPACode}(\text{serializeTerm}(t_2)))$.

2. Por `internalCode_injective`, se obtiene $\text{toSPACode}(\text{serializeTerm}(t_1)) = \text{toSPACode}(\text{serializeTerm}(t_2))$.

3. Por `toSPACode_injective`, se obtiene $\text{serializeTerm}(t_1) = \text{serializeTerm}(t_2)$.

4. Finalmente, por `serializeTerm_injective`, se concluye $t_1 = t_2$.

El mismo razonamiento se aplica a `encodeForm`. La inyectividad de la codificación completa es un resultado técnico profundo: garantiza que la Gödelización es una inyección de la sintaxis del lenguaje objeto en los términos cerrados del nivel inferior. Esta propiedad es indispensable para los argumentos de diagonalización, donde se necesita que la función de codificación sea invertible (al menos inyectiva) para poder construir fórmulas que hablen sobre sus propios códigos de manera no ambigua.

```
/--
   Función de conversión estructural entre el tipo global `_root_.Code`
   y el tipo local `SPA.Code`.
-/
def toSPACode {s : Nat} : _root_.Code s → Code s
  | _root_.Code.nil => Code.nil
  | _root_.Code.cons x xs => Code.cons x (toSPACode xs)

/-- Demostración de que la función de conversión `toSPACode` es inyectiva -/
lemma toSPACode_injective {s : Nat} : Function.Injective (toSPACode (s := s)) := by
  intro c1 c2 h
  induction c1 generalizing c2 with
  | nil =>
    cases c2 with
    | nil => rfl
    | cons y ys => contradiction
  | cons x xs ih =>
    cases c2 with
    | nil => contradiction
    | cons y ys =>
      dsimp [toSPACode] at h
      injection h with hx h_xs
      have h_ys := ih h_xs
      subst hx h_ys
      rfl
```

```
/--
   Definición adaptada de encodeTerm usando el puente de conversión `toSPACode`
   para subsanar la duplicidad de tipos.
-/
def encodeTerm {s : Nat} (t : ArithTerm (s+1)) : TermClosed s :=
  internalCode (toSPACode (serializeTerm t))

/--
   Definición adaptada de encodeForm usando el puente de conversión `toSPACode`
   para subsanar la duplicidad de tipos.
-/
def encodeForm {s : Nat} (φ : ArithForm (s+1)) : TermClosed s :=
  internalCode (toSPACode (serializeForm φ))

/-- Lema de inyectividad para serializeTerm -/
lemma serializeTerm_injective {s : Nat} : Function.Injective (serializeTerm (s := s)) :=
by
  intro t1 t2 h
  have h_dec := congrArg decodeTerm h
  rw [decodeTerm_encodeTerm t1, decodeTerm_encodeTerm t2] at h_dec
  injection h_dec

/-- Lema de inyectividad para serializeForm -/
lemma serializeForm_injective {s : Nat} : Function.Injective (serializeForm (s := s)) :=
by
  intro φ1 φ2 h
  have h_dec := congrArg decodeForm h
  rw [decodeForm_encodeForm φ1, decodeForm_encodeForm φ2] at h_dec
```

```
  injection h_dec

/-- Demostración de la inyectividad de encodeTerm -/
lemma encodeTerm_injective {s : Nat} : Function.Injective (encodeTerm (s := s)) := by
  intro t1 t2 h
  unfold encodeTerm at h
  have h_eq := internalCode_injective h
  have h_ser := toSPACode_injective h_eq
  exact serializeTerm_injective h_ser

/-- Demostración de la inyectividad de encodeForm -/
lemma encodeForm_injective {s : Nat} : Function.Injective (encodeForm (s := s)) := by
  intro φ1 φ2 h
  unfold encodeForm at h
  have h_eq := internalCode_injective h
  have h_ser := toSPACode_injective h_eq
  exact serializeForm_injective h_ser
```

Codificación estructurada de `Code` en el lenguaje objeto

Se define la función `codeToTerm` que traduce un código del metalenguaje (de cualquier estrato superior s_{upper}) a un término aritmético del estrato s, utilizando la codificación numérica provista por la instancia `Encodable` y la conversión a numeral de Peano:

$$ \text{codeToTerm} : \text{Code } s_{\text{upper}} \to \text{ArithTerm } s $$

$$ \text{codeToTerm}(c) := \text{natToTerm}(\text{Encodable.encode}(c)) $$

Intuición y necesidad

Esta función es el análogo simplificado de `internalCode`, pero opera directamente sobre el tipo `Code` sin necesidad de la conversión `toSPACode`. Mientras que `internalCode` produce un `TermClosed s` (un término cerrado empaquetado con su prueba de clausura), `codeToTerm` produce un `ArithTerm s` genérico. Esta versión es más conveniente para construir fórmulas del lenguaje objeto que necesitan incorporar códigos como numerales en posiciones arbitrarias, sin la restricción de que el término resultante deba ser cerrado. La parametrización por dos estratos (s y s_{upper}) permite flexibilidad: el código puede provenir de cualquier nivel superior, pero su representación como numeral siempre reside en el nivel s que se especifique.

Propiedades de preservación para `codeToTerm`

Se demuestran tres propiedades fundamentales que garantizan el buen comportamiento de la codificación:

Lema `isClosed_codeToTerm`: Clausura del término codificado

$$ \forall s, s_{\text{upper}} \in \mathbb{N},\ \forall c \in \text{Code } s_{\text{upper}},\ \text{isClosed}(\text{codeToTerm}_{(s:=s)}\ c) = \text{true} $$

Todo numeral de Peano es inherentemente cerrado, y `codeToTerm` produce precisamente un numeral. La demostración se reduce directamente al lema `isClosed_natToTerm`.

Lema `NotOccur_codeToTerm`: No ocurrencia de variables en el término codificado

$$ \forall s, s_{\text{upper}} \in \mathbb{N},\ \forall c \in \text{Code } s_{\text{upper}},\ \forall v \in \mathbb{N},\ \text{NotOccur } v\ (\text{codeToTerm}_{(s:=s)}\ c) $$

Dado que el término codificado es cerrado, ninguna variable —en particular v— puede ocurrir en él. La demostración invoca el lema auxiliar `NotOccur_of_isClosed`, que establece precisamente que los términos cerrados no contienen ninguna variable.

Lema `codeToTerm_injective`: Inyectividad de la codificación

$$ \forall s, s_{\text{upper}} \in \mathbb{N},\ \forall c_1, c_2 \in \text{Code } s_{\text{upper}},\ \text{codeToTerm}(c_1) = \text{codeToTerm}(c_2) \implies c_1 = c_2 $$

La demostración sigue el mismo patrón que `internalCode_injective`: primero se establece que `natToTerm` es inyectiva (por inducción sobre la estructura de los numerales), luego se deduce $\text{encode}(c_1) = \text{encode}(c_2)$, y finalmente se aplica `Encodable.encode_injective` para concluir $c_1 = c_2$. La inyectividad garantiza que códigos distintos reciben representaciones sintácticas distintas en el lenguaje objeto.

Constructores de fórmulas lógicas derivadas

Se definen los conectivos lógicos clásicos como abreviaturas sintácticas dentro del lenguaje `ArithForm`, utilizando las definiciones estándar a partir de la implicación y la negación:

Conjunción: `and_form`

$$ A \land B := \neg (A \to \neg B) $$

Disyunción: `or_form`

$$ A \lor B := \neg A \to B $$

Cuantificador existencial: `exists_form`

$$ \exists v\ A := \neg \forall v \neg A $$

Intuición y necesidad

Estas definiciones permiten expresar fórmulas más complejas sin extender el tipo inductivo `ArithForm`. La conjunción, disyunción y el cuantificador existencial se definen al estilo de la lógica clásica de primer orden, donde la implicación y la negación (o el cuantificador universal y la negación) forman un conjunto completo de conectivas. Su disponibilidad es indispensable para construir las fórmulas de serialización y diagonalización con una notación más cercana a la práctica matemática estándar.

Subfórmulas de serialización por casos base

Se definen las fórmulas atómicas que capturan cada caso base de la serialización de términos. Para los constructores `var`, `const` y `henkin`, la definición es estructuralmente idéntica (aunque semánticamente distinguible por el contexto):

`φ_var`, `φ_const`, `φ_henkin`

$$ \varphi_{\text{var}} := \exists v_2\ (v_0 = v_2 \land v_1 = v_2) $$

$$ \varphi_{\text{const}} := \exists v_2\ (v_0 = v_2 \land v_1 = v_2) $$

$$ \varphi_{\text{henkin}} := \exists v_2\ (v_0 = v_2 \land v_1 = v_2) $$

Cada una expresa que los valores denotados por v_0 y v_1 son iguales (a través de un testigo existencial v_2). La intención es que estas fórmulas capturen la relación "ambos lados de la ecuación son el mismo código", correspondiente a los casos donde el código serializado tiene una estructura simple.

`φ_zero`

$$ \varphi_{\text{zero}} := (v_0 = \overline{\text{cons}(3, \text{nil})}) \land (v_1 = \overline{\text{cons}(3, \text{nil})}) $$

donde $\overline{\text{cons}(3, \text{nil})}$ es la codificación mediante `codeToTerm` del código correspondiente al constructor `zero` (etiquetado con 3 en el esquema de serialización). Esta fórmula expresa que tanto v_0 como v_1 son exactamente el código del término `zero`.

Intuición y necesidad

Estas subfórmulas son los bloques elementales para construir la fórmula de serialización completa. La estrategia de Gödelización requiere expresar dentro del lenguaje objeto la relación "y es el código de la serialización de x". Esto se descompone en casos según el constructor principal del término x, y cada caso se captura mediante una fórmula específica. Las definiciones presentadas aquí corresponden a los casos base, mientras que los casos recursivos (sucesor, suma, etc.) requerirían definiciones adicionales que involucran la concatenación de códigos.

Fórmula final de serialización para casos base: `φ_ser_base`

Combina las subfórmulas de los casos base en una fórmula única:

$$ \varphi_{\text{ser_base}} := (\varphi_{\text{var}} \lor \varphi_{\text{const}} \lor \varphi_{\text{henkin}} \lor \varphi_{\text{zero}}) \land (v_0 = v_1) $$

Intuición y necesidad

Esta fórmula expresa que v_0 y v_1 satisfacen la relación de serialización para algún caso base, y además son iguales entre sí. La disyunción captura que el término en cuestión corresponde a uno de los cuatro casos base; la conjunción final con $v_0 = v_1$ refuerza la unicidad del código. Esta construcción es un paso preparatorio hacia la definición de una fórmula de serialización completa que cubra todos los constructores de términos, la cual es necesaria para expresar la representabilidad de la función de codificación dentro del sistema formal.

Definiciones adicionales en el tipo `ArithForm`

Se extienden los conectivos derivados como métodos del tipo inductivo `ArithForm`, permitiendo una notación más orientada a objetos:

`ArithForm.exists_form`

$$ \text{ArithForm.exists_form } v\ A := \neg (\forall v \neg A) $$

`ArithForm.and_form`

$$ \text{ArithForm.and_form } A\ B := \neg (A \to \neg B) $$

`lt_form`: Fórmula de orden estricto

$$ x < y := \exists w\ (y = x + S(w)) $$

donde w es una variable fresca que actúa como testigo de la diferencia. Esta fórmula expresa que x es estrictamente menor que y en el sentido de la aritmética de Peano: existe un número positivo $S(w)$ tal que $y = x + S(w)$.

Intuición y necesidad

La definición de `lt_form` es particularmente relevante porque introduce la relación de orden en el lenguaje objeto, la cual es necesaria para expresar propiedades inductivas y acotaciones en las fórmulas de Gödelización. La capacidad de expresar desigualdades dentro del lenguaje formal es indispensable para construir fórmulas

que afirmen, por ejemplo, que una variable es mayor que todas las variables que aparecen en una cierta expresión, o que un código está dentro de un rango específico.

```
--
================================================================
==
-- 1. CODIFICACIÓN ESTRUCTURADA DE CODE EN EL LENGUAJE OBJETO
--
================================================================
==

/-- Traduce recursivamente un Code del metalenguaje a un término de SPA -/
def codeToTerm {s : Nat} {s_upper : Nat} (c : Code s_upper) : ArithTerm s :=
  natToTerm (Encodable.encode c)

-- Propiedades de preservación de clausura y no-ocurrencia para codeToTerm

lemma isClosed_codeToTerm {s : Nat} {s_upper : Nat} (c : Code s_upper) :
    isClosed (codeToTerm (s := s) c) = true :=
  isClosed_natToTerm (Encodable.encode c)

lemma NotOccur_codeToTerm {s : Nat} {s_upper : Nat} (c : Code s_upper) (v : Nat) :
    NotOccur v (codeToTerm (s := s) c) :=
```

```
  NotOccur_of_isClosed (s := s) _ (isClosed_codeToTerm c) v

lemma codeToTerm_injective {s : Nat} {s_upper : Nat} (c1 c2 : Code s_upper)
    (h : codeToTerm (s := s) c1 = codeToTerm (s := s) c2) : c1 = c2 := by
  unfold codeToTerm at h
  have h_nat_inj : ∀ {n m : Nat}, natToTerm (s := s) n = natToTerm m → n = m := by
    intro n m h_eq
    induction n generalizing m with
    | zero =>
      cases m
      · rfl
      · dsimp [natToTerm] at h_eq; injection h_eq
    | succ k ih =>
      cases m
      · dsimp [natToTerm] at h_eq; injection h_eq
      · dsimp [natToTerm] at h_eq; injection h_eq with h_eq'
        rw [ih h_eq']
  have h_encode := h_nat_inj h
  exact Encodable.encode_injective h_encode

--
==========================================================================
==
-- 2. CONSTRUCTORES DE FÓRMULAS LÓGICAS Y CONSTRUCTORES DE CASOS BASE
--
==========================================================================
==

def and_form {s : Nat} (A B : ArithForm s) : ArithForm s :=
  ArithForm.neg (ArithForm.impl A (ArithForm.neg B))
```

```
def or_form {s : Nat} (A B : ArithForm s) : ArithForm s :=
  ArithForm.impl (ArithForm.neg A) B

def exists_form {s : Nat} (v : Nat) (A : ArithForm s) : ArithForm s :=
  ArithForm.neg (ArithForm.forall_form v (ArithForm.neg A))

-- Definición de las subfórmulas de serialización por casos

def φ_var {s : Nat} : ArithForm s :=
  exists_form 2 (and_form (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 2))
                 (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 2)))

def φ_const {s : Nat} : ArithForm s :=
  exists_form 2 (and_form (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 2))
                 (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 2)))

def φ_henkin {s : Nat} : ArithForm s :=
  exists_form 2 (and_form (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 2))
                 (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 2)))

def φ_zero {s : Nat} : ArithForm s :=
  and_form (ArithForm.eq (ArithTerm.var 0) (codeToTerm (Code.cons (s := s + 1) 3
(Code.nil (s := s + 1)))))
        (ArithForm.eq (ArithTerm.var 1) (codeToTerm (Code.cons (s := s + 1) 3
(Code.nil (s := s + 1)))))

-- Fórmula final de serialización (casos base)
```

```
def φ_ser_base {s : Nat} : ArithForm s :=
  and_form (or_form φ_var (or_form φ_const (or_form φ_henkin φ_zero)))
        (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 1))



/-- Cuantificador existencial definido sobre el universal -/
def ArithForm.exists_form {s : Nat} (v : Nat) (A : ArithForm s) : ArithForm s :=
  ArithForm.neg (ArithForm.forall_form v (ArithForm.neg A))

/-- Conjunción lógica definida sobre implicación y negación -/
def ArithForm.and_form {s : Nat} (A B : ArithForm s) : ArithForm s :=
  ArithForm.neg (ArithForm.impl A (ArithForm.neg B))

/-- Fórmula x < y, usando la variable w como testigo existencial -/
def lt_form {s : Nat} (x y : ArithTerm s) (w : Nat) : ArithForm s :=
  ArithForm.exists_form w
    (ArithForm.eq y (ArithTerm.add x (ArithTerm.succ (ArithTerm.var w))))
```

Función β de Gödel en el metalenguaje

Se define la función β de Gödel, una función aritmética triádica que constituye el núcleo técnico del lema de representabilidad de secuencias finitas:

$$ \beta(a, b, i) := a \bmod (1 + (i+1) \cdot b) $$

Intuición y necesidad

La función β de Gödel es la herramienta fundamental para codificar secuencias finitas de números naturales mediante pares de números. La idea central es que para cualquier secuencia finita $(x_0, \ldots, x_{n-1})$ de números naturales, existen $a, b \in \mathbb{N}$ tales que $\beta(a, b, i) = x_i$ para todo $i < n$. Esto se logra explotando el teorema chino del resto: tomando b como un número divisible por todos los enteros $1, 2, \ldots, n$, y a como la solución simultánea de las congruencias $a \equiv x_i \pmod{1 + (i+1)b}$. La función β permite así reducir la noción de secuencia finita —que es intrínsecamente de segundo orden— a una relación aritmética de primer orden, lo cual es indispensable para la aritmetización de la sintaxis en las pruebas de incompletitud.

Fórmula objeto `BetaForm`

Se define la fórmula del lenguaje objeto que expresa la relación $r = \beta(a, b, i)$ dentro del sistema SPA:

$$ \text{BetaForm} := \exists v_4\ (a = v_4 \cdot M + r \land r < M) $$

donde:

- a es la variable v_0
- b es la variable v_1
- i es la variable v_2
- r es la variable v_3
- $M := S((S(i)) \cdot b) = 1 + (i+1) \cdot b$

Formalmente:

$$ \text{BetaForm}(a, b, i, r) \equiv \exists w\ (a = w \cdot (1 + (i+1) \cdot b) + r \land r < 1 + (i+1) \cdot b) $$

Intuición y necesidad

`BetaForm` es la internalización de la función β dentro del lenguaje formal. La fórmula expresa que r es el resto de dividir a por $1 + (i+1) \cdot b$, lo cual equivale exactamente a $r = a \bmod (1 + (i+1) \cdot b)$. La variable v_4 actúa como testigo existencial del cociente w en la división euclídea. La condición $r < M$ garantiza que r es efectivamente el resto (y no un número congruente mayor). Esta fórmula es un bloque constructivo esencial para expresar propiedades sobre secuencias finitas dentro del lenguaje de la aritmética, y por ende para la Gödelización de nociones como "ser una prueba" o "ser una fórmula".

Fórmula `PairForm`: Emparejamiento de Gödel

Se define la fórmula que representa la función de emparejamiento canónica de Gödel en el lenguaje de SPA, con exactamente tres variables libres (v_0, v_1, v_2):

$$ \text{PairForm}(x, y, p) := (x < y \land p = y^2 + x) \lor (\neg(x < y) \land p = x^2 + x + y) $$

donde v_3 se utiliza como variable ligada interna para el testigo de `lt_form`. Formalmente:

$$ \text{PairForm} := \text{or_form}\ (\text{and_form}\ (\text{lt_form}(v_0, v_1, 3))\ (v_2 = v_1^2 + v_0))\ (\text{and_form}\ (\neg \text{lt_form}(v_0, v_1, 3))\ (v_2 = v_0^2 + v_0 + v_1)) $$

Intuición y necesidad

La función de emparejamiento de Gödel es una biyección computable entre $\mathbb{N} \times \mathbb{N}$ y $\mathbb{N}$ definida por:

$$ \langle x, y \rangle = \begin{cases} y^2 + x & \text{si } x < y \\ x^2 + x + y & \text{si } x \geq y \end{cases} $$

`PairForm` internaliza esta función en el lenguaje objeto. La distinción por casos según $x < y$ o $x \geq y$ asegura la inyectividad de la codificación: cada par ordenado recibe un único código. Esta fórmula es indispensable para la Gödelización porque permite reducir relaciones binarias (como "el término t se sustituye en la fórmula φ") a propiedades unarias sobre códigos, allanando el camino para la representabilidad de todas las nociones sintácticas recursivas.

Lemas de derivación interna de reglas lógicas en SPA

Se demuestra que los conectivos derivados (`exists_form`, `or_form`, `and_form`) satisfacen las reglas de introducción y eliminación habituales dentro del sistema deductivo interno `SPA_Deduction_Internal`. Estos lemas son necesarios para razonar sobre deducciones que involucran las fórmulas de Gödelización sin tener que expandir manualmente las definiciones cada vez.

Lema `exists_intro`: Introducción del existencial

Establece la regla de introducción del cuantificador existencial: si se puede deducir $\varphi[v := t]$ y el término t es libre para v en φ, entonces se puede deducir $\exists v\ \varphi$:

$$ \forall \Gamma, v, \varphi, t,\ \text{IsFreeFor}(v, t, \varphi) \land \Gamma \vdash \varphi[v := t] \implies \Gamma \vdash \exists v\ \varphi $$

Idea de la demostración

Se procede por reducción al absurdo utilizando la definición de $\exists$ como $\neg \forall \neg$. Se supone $\forall v \neg \varphi$ y se aplica la eliminación del

universal con el término t (lo cual es legítimo por la condición `IsFreeFor`) para obtener $\neg \varphi[v := t]$. Esto contradice la hipótesis $\Gamma \vdash \varphi[v := t]$, por lo que se concluye $\neg \forall v \neg \varphi$, es decir, $\exists v\ \varphi$.

Lemas `or_intro_left` y `or_intro_right`: Introducción de la disyunción

$$ \Gamma \vdash A \implies \Gamma \vdash A \lor B $$

$$ \Gamma \vdash B \implies \Gamma \vdash A \lor B $$

Idea de la demostración

Para `or_intro_left`, se utiliza la definición $A \lor B := \neg A \to B$. Se aplica introducción de la implicación: en el contexto extendido con $\neg A$, se tiene tanto $\neg A$ (por hipótesis) como A (por debilitamiento desde Γ), lo que produce una contradicción y de ella se deriva B por el principio ex falso quodlibet. `or_intro_right` es más directo: en el contexto extendido con $\neg A$, simplemente se debilita la prueba de B desde Γ.

Lema `and_intro`: Introducción de la conjunción

$$ \Gamma \vdash A \land \Gamma \vdash B \implies \Gamma \vdash A \land B $$

Idea de la demostración

Recordando que $A \land B := \neg (A \to \neg B)$, se procede por introducción de la negación. Se supone $A \to \neg B$ en el contexto extendido. Por debilitamiento se tienen A y B desde Γ. Por eliminación de la implicación se obtiene $\neg B$, que contradice B. Por tanto, se niega la suposición, obteniendo $\neg (A \to \neg B)$.

Lema `and_elim_right`: Eliminación derecha de la conjunción

$$ \Gamma \vdash A \land B \implies \Gamma \vdash B $$

Idea de la demostración

Se requiere demostrar B a partir de $\neg (A \to \neg B)$. Se aplica doble negación para reducir el problema a demostrar $\neg \neg B$. Se supone $\neg B$ y se construye una prueba de $A \to \neg B$ (por introducción de la implicación, ya que $\neg B$ está disponible como hipótesis). Esto contradice $\neg (A \to \neg B)$, que se tiene por debilitamiento desde Γ. La contradicción fuerza $\neg \neg B$, y por doble negación se obtiene B.

--
==========
==

-- 3. LEMAS DE DERIVACIÓN INTERNA DE REGLAS LÓGICAS EN SPA

--
==========
==

/-- 1. Función β de Gödel en el metalenguaje -/

def godelBeta (a b i : Nat) : Nat := a % (1 + (i+1)*b)

/-- 1. Fórmula objeto BetaForm (Línea 8324 del archivo) -/

```
def BetaForm {s : Nat} : ArithForm s :=
  let a := ArithTerm.var 0
  let b := ArithTerm.var 1
  let i := ArithTerm.var 2
  let r := ArithTerm.var 3
  let M := ArithTerm.succ (ArithTerm.mul (ArithTerm.succ i) b)
  exists_form 4
    (and_form
      (ArithForm.eq a (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) M) r))
      (lt_form r M 5))
```

```
--
================================================================
-- DEFINICIÓN DE PAIRFORM
--
================================================================
```

```
/--
  Fórmula que representa la función de emparejamiento de Gödel en el lenguaje de SPA.
  Tiene exactamente las variables libres 0, 1 y 2.
  Utiliza la variable ligada interna 3 para el testigo de lt_form.
-/
def PairForm {s : Nat} : ArithForm s :=
```

```
  or_form
    (and_form (lt_form (ArithTerm.var 0) (ArithTerm.var 1) 3)
          (ArithForm.eq (ArithTerm.var 2) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 1) (ArithTerm.var 1)) (ArithTerm.var 0))))
    (and_form (ArithForm.neg (lt_form (ArithTerm.var 0) (ArithTerm.var 1) 3))
          (ArithForm.eq (ArithTerm.var 2) (ArithTerm.add (ArithTerm.add (ArithTerm.mul (ArithTerm.var 0) (ArithTerm.var 0)) (ArithTerm.var 0)) (ArithTerm.var 1))))

lemma exists_intro {s : Nat} (Γ : Set (ArithForm s)) (v : Nat) (φ : ArithForm s) (t : ArithTerm s)
    (h_free : IsFreeFor v t φ) (h : SPA_Deduction_Internal s Γ (substForm v t φ)) :
    SPA_Deduction_Internal s Γ (exists_form v φ) := by
  apply SPA_Deduction_Internal.neg_intro
  have h_forall : SPA_Deduction_Internal s (insert (ArithForm.forall_form v (ArithForm.neg φ)) Γ) (ArithForm.forall_form v (ArithForm.neg φ)) :=
    SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
  have h_elim := SPA_Deduction_Internal.forall_elim _ v (ArithForm.neg φ) t h_forall h_free
  rw [substForm_neg] at h_elim
  have h_weak : SPA_Deduction_Internal s (insert (ArithForm.forall_form v (ArithForm.neg φ)) Γ) (substForm v t φ) :=
    SPA_Deduction_Internal.weaken Γ _ _ h (Set.subset_insert _ _)
  exact SPA_Deduction_Internal.neg_elim _ _ h_weak h_elim

lemma or_intro_left {s : Nat} (Γ : Set (ArithForm s)) (A B : ArithForm s)
```

```
    (h : SPA_Deduction_Internal s Γ A) :
    SPA_Deduction_Internal s Γ (or_form A B) := by
  apply SPA_Deduction_Internal.imp_intro
  have h_negA : SPA_Deduction_Internal s (insert (ArithForm.neg A) Γ) (ArithForm.neg
A) :=
    SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
  have h_A : SPA_Deduction_Internal s (insert (ArithForm.neg A) Γ) A :=
    SPA_Deduction_Internal.weaken Γ _ _ h (Set.subset_insert _ _)
  have h_bot := SPA_Deduction_Internal.neg_elim _ _ h_A h_negA
  exact ex_falso _ B h_bot

lemma or_intro_right {s : Nat} (Γ : Set (ArithForm s)) (A B : ArithForm s)
    (h : SPA_Deduction_Internal s Γ B) :
    SPA_Deduction_Internal s Γ (or_form A B) := by
  apply SPA_Deduction_Internal.imp_intro
  exact SPA_Deduction_Internal.weaken Γ _ _ h (Set.subset_insert _ _)

lemma and_intro {s : Nat} (Γ : Set (ArithForm s)) (A B : ArithForm s)
    (hA : SPA_Deduction_Internal s Γ A) (hB : SPA_Deduction_Internal s Γ B) :
    SPA_Deduction_Internal s Γ (and_form A B) := by
  apply SPA_Deduction_Internal.neg_intro
  have h_imp : SPA_Deduction_Internal s (insert (ArithForm.impl A (ArithForm.neg B))
Γ) (ArithForm.impl A (ArithForm.neg B)) :=
    SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
  have h_A : SPA_Deduction_Internal s (insert (ArithForm.impl A (ArithForm.neg B)) Γ)
A :=
    SPA_Deduction_Internal.weaken Γ _ _ hA (Set.subset_insert _ _)
  have h_negB := SPA_Deduction_Internal.imp_elim _ _ _ h_imp h_A
  have h_B : SPA_Deduction_Internal s (insert (ArithForm.impl A (ArithForm.neg B)) Γ)
B :=
```

```
    SPA_Deduction_Internal.weaken Γ _ _ hB (Set.subset_insert _ _)
  exact SPA_Deduction_Internal.neg_elim _ _ h_B h_negB

lemma and_elim_right {s : Nat} (Γ : Set (ArithForm s)) (A B : ArithForm s)
    (h : SPA_Deduction_Internal s Γ (and_form A B)) :
    SPA_Deduction_Internal s Γ B := by
  apply SPA_Deduction_Internal.double_neg
  apply SPA_Deduction_Internal.neg_intro
  have h_imp : SPA_Deduction_Internal s (insert (ArithForm.neg B) Γ) (ArithForm.impl A (ArithForm.neg B)) := by
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.hyp
    exact Or.inr (Or.inl rfl)
  have h_neg_imp : SPA_Deduction_Internal s (insert (ArithForm.neg B) Γ) (ArithForm.neg (ArithForm.impl A (ArithForm.neg B))) :=
    SPA_Deduction_Internal.weaken Γ _ _ h (Set.subset_insert _ _)
  exact SPA_Deduction_Internal.neg_elim _ _ h_imp h_neg_imp
```

Lemas de soporte de no-ocurrencia para las subfórmulas de serialización

Se demuestra que las subfórmulas de serialización para los casos base (`φ_var`, `φ_const`, `φ_henkin`, `φ_zero`) no contienen variables libres distintas de v_0 y v_1. Formalmente, para cada una de estas fórmulas φ y para cualquier variable v que no sea 0 ni 1:

$$ \forall s \in \mathbb{N},\ \forall v \in \mathbb{N},\ v \neq 0 \land v \neq 1 \implies \text{NotFreeForm } v\ (\varphi_{(s:=s)}) $$

Lemas `NotFreeForm_φ_var`, `NotFreeForm_φ_const`, `NotFreeForm_φ_henkin`

Para las tres primeras subfórmulas, la estructura de la demostración es idéntica debido a su definición estructuralmente análoga. Cada una se define como:

$$ \varphi_{\text{var}} := \exists v_2\ (v_0 = v_2 \land v_1 = v_2) $$

La demostración analiza si $v = 2$:

- **Caso $v = 2$:** La variable v_2 está ligada por el cuantificador existencial, por lo que no ocurre libre en la fórmula. La definición de `NotFreeForm` para `exists_form` establece que si v es la variable ligada, entonces v no es libre (se satisface trivialmente).

- **Caso $v \neq 2$:** Se despliegan las definiciones de `exists_form`, `and_form` y `NotFreeForm`. La conjunción requiere que v no ocurra libre en ninguna de las dos igualdades. Cada igualdad involucra las variables 0 (o 1) y 2. Dado que $v \neq 0$, $v \neq 1$ y $v \neq 2$, se concluye que v no ocurre en ningún término de las igualdades mediante el constructor `NotOccur.var`, que establece que una variable v no ocurre en una variable m cuando $v \neq m$.

Intuición y necesidad

Estos lemas certifican que φ_{var}, φ_{const} y φ_{henkin} tienen exactamente dos variables libres: v_0 y v_1. Esta propiedad es indispensable para que estas fórmulas puedan ser utilizadas como la fórmula representadora φ en la definición de `IsRepresentable`, donde se exige que solo v_0 y v_1 ocurran libres. Cualquier variable adicional que apareciera libre introduciría parámetros ocultos que invalidarían la representabilidad.

Lema `NotFreeForm_φ_zero`

Para la subfórmula `φ_zero`, la situación es ligeramente diferente debido a la presencia de `codeToTerm`:

$$ \varphi_{\text{zero}} := (v_0 = \overline{c}) \land (v_1 = \overline{c}) $$

donde $c = \text{cons}(3, \text{nil})$. La demostración descompone la conjunción en sus dos componentes. Para cada igualdad, se requiere demostrar que v no ocurre en el lado izquierdo (que es una variable) ni en el lado derecho (que es un numeral codificado):

- **Lado izquierdo:** Dado que $v \neq 0$ (respectivamente $v \neq 1$), se tiene $\text{NotOccur.var } 0$ (respectivamente $\text{NotOccur.var } 1$).

- **Lado derecho:** El lema `NotOccur_codeToTerm` establece que ninguna variable ocurre en un término producido por `codeToTerm`, ya que este genera numerales cerrados. En particular, v no ocurre en $\overline{c}$.

Intuición y necesidad

La verificación de que φ_{zero} solo tiene libres a v_0 y v_1 es crucial para la coherencia de la fórmula de serialización completa. A diferencia de las otras subfórmulas, `φ_zero` incorpora términos codificados mediante `codeToTerm`, lo cual introduce objetos sintácticos potencialmente complejos. La propiedad de clausura de `codeToTerm` (demostrada en `isClosed_codeToTerm` y refinada en `NotOccur_codeToTerm`) garantiza que estos numerales no introducen variables libres no deseadas, preservando así la pureza de la fórmula respecto a sus variables libres declaradas.

```
--
==========================================================================
==
-- 4. LEMAS DE SOPORTE DE NO-OCURRENCIA PARA LAS SUBFÓRMULAS
--
==========================================================================
==

lemma NotFreeForm_φ_var {s : Nat} (v : Nat) (hv0 : v ≠ 0) (hv1 : v ≠ 1) :
   NotFreeForm v (φ_var (s := s)) := by
 dsimp [φ_var, exists_form, and_form, NotFreeForm]
 by_cases hv2 : v = 2
 · subst hv2; trivial
 · rw [if_neg hv2]
   exact ⟨⟨NotOccur.var 0 hv0, NotOccur.var 2 hv2⟩, ⟨NotOccur.var 1 hv1, NotOccur.var
2 hv2⟩⟩

lemma NotFreeForm_φ_const {s : Nat} (v : Nat) (hv0 : v ≠ 0) (hv1 : v ≠ 1) :
   NotFreeForm v (φ_const (s := s)) := by
 dsimp [φ_const, exists_form, and_form, NotFreeForm]
 by_cases hv2 : v = 2
```

```
  · subst hv2; trivial
  · rw [if_neg hv2]
    exact ⟨⟨NotOccur.var 0 hv0, NotOccur.var 2 hv2⟩, ⟨NotOccur.var 1 hv1, NotOccur.var
2 hv2⟩⟩

lemma NotFreeForm_φ_henkin {s : Nat} (v : Nat) (hv0 : v ≠ 0) (hv1 : v ≠ 1) :
    NotFreeForm v (φ_henkin (s := s)) := by
  dsimp [φ_henkin, exists_form, and_form, NotFreeForm]
  by_cases hv2 : v = 2
  · subst hv2; trivial
  · rw [if_neg hv2]
    exact ⟨⟨NotOccur.var 0 hv0, NotOccur.var 2 hv2⟩, ⟨NotOccur.var 1 hv1, NotOccur.var
2 hv2⟩⟩

lemma NotFreeForm_φ_zero {s : Nat} (v : Nat) (hv0 : v ≠ 0) (hv1 : v ≠ 1) :
    NotFreeForm v (φ_zero (s := s)) := by
  unfold φ_zero and_form NotFreeForm
  constructor
  · constructor
    · exact NotOccur.var 0 (by omega)
    · exact NotOccur_codeToTerm (Code.cons (s := s+1) 3 (Code.nil (s := s+1))) v
  · constructor
    · exact NotOccur.var 1 (by omega)
    · exact NotOccur_codeToTerm (Code.cons (s := s+1) 3 (Code.nil (s := s+1))) v
```

Lemas de equivalencia para `encodeTerm` en cada caso base

Se establecen las igualdades explícitas que caracterizan el comportamiento de la función de codificación `encodeTerm` sobre cada constructor base de términos aritméticos. Estos lemas descomponen la codificación en sus componentes elementales, mostrando exactamente qué código numérico corresponde a cada caso sintáctico.

Lema `encodeTerm_var`: Codificación de una variable

$$ \forall s \in \mathbb{N},\ \forall n \in \mathbb{N},\ (\text{encodeTerm}_{(s:=s)}(\text{var } n)).\text{val} = \overline{\text{cons}(0, \text{cons}(n, \text{nil}))} $$

donde $\overline{\cdot}$ denota la traducción mediante `codeToTerm`. El código comienza con 0 (etiqueta del constructor `var`), seguido del índice n de la variable.

Lema `encodeTerm_const`: Codificación de una constante numérica

$$ \forall s \in \mathbb{N},\ \forall c \in \text{NatS}(s+1),\ (\text{encodeTerm}_{(s:=s)}(\text{const } c)).\text{val} = \overline{\text{cons}(1, \text{cons}(\text{encode}(c), \text{nil}))} $$

El código comienza con 1 (etiqueta del constructor `const`), seguido del código numérico de la constante sintáctica c mediante `NatS_encode`.

Lema `encodeTerm_henkin`: Codificación de una constante de Henkin

$$ \forall s \in \mathbb{N},\ \forall h \in \text{HenkinConst},\ (\text{encodeTerm}_{(s:=s)}(\text{henkin } h)).\text{val} = \overline{\text{cons}(2, \text{cons}(h.\text{index}, \text{nil}))} $$

El código comienza con 2 (etiqueta del constructor `henkin`), seguido del índice de la constante de Henkin h.

Lema `encodeTerm_zero`: Codificación del cero aritmético

$$ \forall s \in \mathbb{N},\ (\text{encodeTerm}_{(s:=s)}(\text{zero})).\text{val} = \overline{\text{cons}(3, \text{nil})} $$

El código consiste únicamente en la etiqueta 3 correspondiente al constructor `zero`, sin argumentos adicionales.

Intuición y necesidad

Estos lemas hacen explícita la correspondencia entre la sintaxis del lenguaje objeto y su representación numérica, materializando el esquema de Gödelización definido previamente por `serializeTerm`. Las etiquetas numéricas (0 para variables, 1 para constantes, 2 para Henkin, 3 para cero, 4 para sucesor, etc.) constituyen un código libre de ambigüedad que permite recuperar unívocamente la estructura del término original a partir de su código.

La importancia de estos lemas radica en que permiten razonar sobre la codificación dentro del sistema deductivo mismo. Cuando se construyen fórmulas que afirman propiedades sobre códigos (por ejemplo, "el código x corresponde a una variable"), estas igualdades proporcionan la base para verificar que las subfórmulas de serialización (`φ_var`, `φ_zero`, etc.) capturan correctamente cada caso. En particular, nótese la coincidencia entre el código de `zero` ($\text{cons}(3, \text{nil})$) y el término utilizado en la definición de `φ_zero`, lo cual valida que dicha fórmula caracteriza exactamente los códigos que provienen del término `zero`. La igualdad es por definición (`rfl`), lo que refleja que la implementación es correcta por

construcción: la serialización, la conversión de tipos y la codificación interna se han diseñado para que estas ecuaciones se satisfagan trivialmente.

```
--
==========================================================================
==

-- 5. LEMAS DE EQUIVALENCIA PARA ENCODETERM EN CADA CASO BASE

--
==========================================================================
==


lemma encodeTerm_var {s : Nat} (n : Nat) :

  (encodeTerm (s := s) (ArithTerm.var n)).val = codeToTerm (Code.cons (s := s + 1) 0
(Code.cons (s := s + 1) n (Code.nil (s := s + 1)))) := rfl


lemma encodeTerm_const {s : Nat} (c : NatS (s+1)) :

  (encodeTerm (s := s) (ArithTerm.const c)).val = codeToTerm (Code.cons (s := s + 1)
1 (Code.cons (s := s + 1) (NatS_encode c) (Code.nil (s := s + 1)))) := rfl


lemma encodeTerm_henkin {s : Nat} (h : HenkinConst) :

  (encodeTerm (s := s) (ArithTerm.henkin h)).val = codeToTerm (Code.cons (s := s +
1) 2 (Code.cons (s := s + 1) h.index (Code.nil (s := s + 1)))) := rfl


lemma encodeTerm_zero {s : Nat} :

  (encodeTerm (s := s) ArithTerm.zero).val = codeToTerm (Code.cons (s := s + 1) 3
(Code.nil (s := s + 1))) := rfl
```

Demostración de la representabilidad de `serializeTerm` para los casos base

Se demuestra el teorema central que establece que la función de serialización de términos, restringida a los casos base (variables, constantes numéricas, constantes de Henkin y cero), es representable en cualquier teoría Δ maximalmente consistente y con propiedad de Henkin. Formalmente:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } } s,\ \text{Maximal}(\Delta) \land \text{HenkinSet}(\Delta) \land \text{Consistent}(\Delta) \implies \exists \varphi \in \text{ArithForm } } s, $$

tal que φ satisface las condiciones de `IsRepresentable` para la función que mapea cada término base a su código interno:

$$ f(t) := \text{internalCode}(\text{toSPACode}(\text{serializeTerm}(t))) $$

Estructura de la demostración

La prueba utiliza `φ_ser_base` como la fórmula representadora y verifica las cinco condiciones requeridas por `IsRepresentable`:

1. La variable v_0 es libre en `φ_ser_base`: Se demuestra por contradicción: si v_0 no fuera libre, la definición de `NotFreeForm` forzaría que v_0 no ocurre en las subfórmulas, pero `φ_ser_base` contiene explícitamente v_0 en la igualdad final y en las subfórmulas disyuntivas.

2. La variable v_1 es libre en `φ_ser_base`: Análogo al caso anterior, utilizando que v_1 aparece en la igualdad final de la fórmula.

3. Ninguna otra variable es libre: Para cualquier $v \neq 0, 1$, se invocan los lemas `NotFreeForm_φ_var`, `NotFreeForm_φ_const`, `NotFreeForm_φ_henkin` y `NotFreeForm_φ_zero`, que garantizan que v no ocurre libre en ninguna de las subfórmulas. La conjunción y la disyunción preservan esta propiedad, por lo que v no es libre en `φ_ser_base`.

---

4. Condición de gráfico: Para cada caso base, se debe demostrar que la instancia adecuada de `φ_ser_base` pertenece a Δ. La estrategia es uniforme para los cuatro casos:

- Se reescribe el código de `encodeTerm` usando los lemas `encodeTerm_var`, `encodeTerm_const`, `encodeTerm_henkin` o `encodeTerm_zero`.

- Se define w como el numeral que codifica el término correspondiente.

- La fórmula instanciada tiene la estructura:

$$ (\text{disyunción de casos})(w, w) \land (w = w) $$

- Para la parte izquierda (la disyunción), se demuestra que la subfórmula correspondiente al caso específico pertenece a Δ, utilizando que w es un término cerrado y aplicando introducción del existencial (`exists_intro`) sobre la conjunción $w = w \land w = w$, que a su vez pertenece a Δ por el lema `and_self_in_Δ`.

- Mediante aplicaciones sucesivas de `or_intro_left` y `or_intro_right`, se propaga la pertenencia a la disyunción completa.

- Para la parte derecha ($w = w$), se utiliza el lema `eq_refl_in_Δ`.

- Finalmente, `and_intro` combina ambas partes y se cierra bajo deducción en Δ.

---

5. Condición de funcionalidad: Para cada caso base y cualquier término cerrado c tal que la instancia de `φ_ser_base` con c pertenece a Δ, se debe demostrar $c = \text{encodeTerm}(t) \in \Delta$. La estrategia es:

- Se reescribe la hipótesis h_{in} utilizando el lema de codificación correspondiente.

- La fórmula `φ_ser_base` instanciada es una conjunción cuya segunda componente es exactamente la igualdad $c = w$ (donde w es el código del término base), gracias al lema `substTerm_fresh` que garantiza que la sustitución de v_1 por c en $v_0 = v_1$ produce $c = c$, pero la estructura de `φ_ser_base` fuerza que v_0 y v_1 se sustituyan por w y c respectivamente, resultando en $c = w$.

- Se aplica `and_elim_right` para extraer esta igualdad de la conjunción.

- Por simetría de la igualdad (`eq_symm`), se obtiene $w = c$, que es exactamente $\text{encodeTerm}(t).\text{val} = c.\text{val}$.

Intuición y necesidad

Este teorema constituye la primera etapa de la demostración de representabilidad de la función de serialización completa. Al restringirse a los casos base, se establece el patrón de razonamiento que deberá extenderse a los casos recursivos (sucesor, suma, producto, etc.) en desarrollos posteriores. La construcción revela la estructura profunda de la Gödelización: la fórmula representadora `φ_ser_base` actúa como un "programa lógico" que, mediante disyunciones anidadas, verifica a qué caso sintáctico corresponde el código de entrada y fuerza la igualdad con el código de salida. La maximalidad y consistencia de Δ garantizan que las deducciones internas se reflejan fielmente en la pertenencia a Δ, mientras que la propiedad de Henkin (aunque no utilizada explícitamente en los casos base) será indispensable para los casos existenciales en la extensión recursiva.

```
--
============================================================
==
-- 6. DEMOSTRACIÓN DE LA REPRESENTABILIDAD DE SERIALIZERTERM
--
============================================================
==


lemma eq_refl_in_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ) (w : ArithTerm s) :

  ArithForm.eq w w ∈ Δ := by

 apply maximal_consistent_closed_under_deduction Δ h_max h_con

 apply SPA_Deduction_Internal.eq_refl


lemma and_self_in_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ) (w : ArithTerm s) :

  and_form (ArithForm.eq w w) (ArithForm.eq w w) ∈ Δ := by

 have h_eq := eq_refl_in_Δ Δ h_max h_con w

 have h_eq_ded := SPA_Deduction_Internal.hyp Δ _ h_eq

 have h_and := and_intro Δ _ _ h_eq_ded h_eq_ded

 exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and


lemma serializeTerm_base_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max :
Maximal Δ)

  (_h_henkin : HenkinSet Δ) (h_con : Consistent Δ) :

  ∃ (φ_ser_base_formula : ArithForm s)

   (_h_v0 : ¬ NotFreeForm 0 φ_ser_base_formula)

   (_h_v1 : ¬ NotFreeForm 1 φ_ser_base_formula)

   (_h_only : ∀ (v : Nat), v ≠ 0 → v ≠ 1 → NotFreeForm v φ_ser_base_formula),

   -- Gráfico para los casos base

   (∀ (t : ArithTerm (s+1)),
```

```
    (∃ (n : Nat), t = ArithTerm.var n) ∨
    (∃ (c : NatS (s+1)), t = ArithTerm.const c) ∨
    (∃ (h : HenkinConst), t = ArithTerm.henkin h) ∨
    (t = ArithTerm.zero) →
    let t_code := encodeTerm t
    let c := internalCode (toSPACode (serializeTerm t))
    substForm 1 c.val (substForm 0 t_code.val φ_ser_base_formula) ∈ Δ) ∧
  -- Funcionalidad para los casos base
  (∀ (t : ArithTerm (s+1)) (c : TermClosed s),
    ((∃ (n : Nat), t = ArithTerm.var n) ∨
     (∃ (c' : NatS (s+1)), t = ArithTerm.const c') ∨
     (∃ (h : HenkinConst), t = ArithTerm.henkin h) ∨
     (t = ArithTerm.zero)) →
    substForm 1 c.val (substForm 0 (encodeTerm t).val φ_ser_base_formula) ∈ Δ →
    ArithForm.eq c.val (internalCode (toSPACode (serializeTerm t))).val ∈ Δ) := by
use φ_ser_base
refine ⟨?_, ?_, ?_, ⟨?_, ?_⟩⟩
· -- v₀ es libre en la fórmula (no es no libre)
  intro h
  dsimp [φ_ser_base, and_form, NotFreeForm] at h
  rcases h with ⟨_, h0, _⟩
  cases h0 with
  | var m h_ne =>
    exact h_ne rfl
· -- v₁ es libre en la fórmula (no es no libre)
  intro h
  dsimp [φ_ser_base, and_form, NotFreeForm] at h
  rcases h with ⟨_, _, h1⟩
```

cases h1 with

| var m h_ne =>

exact h_ne rfl

· -- Ninguna otra variable aparte de v_0 y v_1 es libre

intro v hv0 hv1

dsimp [φ_ser_base, and_form, NotFreeForm]

refine ⟨?_, NotOccur.var 0 (by omega), NotOccur.var 1 (by omega)⟩

dsimp [or_form, NotFreeForm]

refine ⟨NotFreeForm_φ_var v hv0 hv1, ?_⟩

refine ⟨NotFreeForm_φ_const v hv0 hv1, ?_⟩

exact ⟨NotFreeForm_φ_henkin v hv0 hv1, NotFreeForm_φ_zero v hv0 hv1⟩

· -- Condición de Gráfico para los casos base

intro t h_base

change substForm 1 (encodeTerm t).val (substForm 0 (encodeTerm t).val φ_ser_base) ∈ Δ

rcases h_base with ⟨n, rfl⟩ | ⟨c, rfl⟩ | ⟨h, rfl⟩ | rfl

· -- Caso var n

rw [encodeTerm_var]

let w := codeToTerm (s := s) (Code.cons (s := s+1) 0 (Code.cons (s := s+1) n (Code.nil (s := s+1))))

have h_closed : isClosed w = true := isClosed_codeToTerm (s := s) (Code.cons (s := s+1) 0 (Code.cons (s := s+1) n (Code.nil (s := s+1))))

let A := substForm 1 w (substForm 0 w φ_var)

let B := substForm 1 w (substForm 0 w φ_const)

let C := substForm 1 w (substForm 0 w φ_henkin)

let D := substForm 1 w (substForm 0 w φ_zero)

change and_form (or_form A (or_form B (or_form C D))) (ArithForm.eq (substTerm 1 w w) (substTerm 1 w (ArithTerm.var 1))) ∈ Δ

rw [substTerm_fresh w 1 w (NotOccur_of_isClosed w h_closed 1)]

have h_cases_subst : or_form A (or_form B (or_form C D)) ∈ Δ := by

have h_var_subst : A ∈ Δ := by

change exists_form 2 (and_form (ArithForm.eq (substTerm 1 w w) (ArithTerm.var 2)) (ArithForm.eq w (ArithTerm.var 2))) ∈ Δ

rw [substTerm_fresh w 1 w (NotOccur_of_isClosed w h_closed 1)]

have h_eq_and := and_self_in_Δ Δ h_max h_con w

have h_eq_and_ded := SPA_Deduction_Internal.hyp Δ _ h_eq_and

let F := and_form (ArithForm.eq w (ArithTerm.var 2)) (ArithForm.eq w (ArithTerm.var 2))

have h_free : IsFreeFor 2 w F := IsFreeFor_closed F 2 ⟨w, h_closed⟩

have h_subst_F : substForm 2 w F = and_form (ArithForm.eq (substTerm 2 w w) w) (ArithForm.eq (substTerm 2 w w) w) := rfl

have h_eq_and_ded' : SPA_Deduction_Internal s Δ (substForm 2 w F) := by

rw [h_subst_F]

rw [substTerm_fresh w 2 w (NotOccur_of_isClosed w h_closed 2)]

exact h_eq_and_ded

have h_exists := exists_intro Δ 2 F w h_free h_eq_and_ded'

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists

have h_or_ded := or_intro_left Δ A (or_form B (or_form C D)) (SPA_Deduction_Internal.hyp Δ _ h_var_subst)

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_or_ded

have h_cases_subst' : substForm 1 w (substForm 0 w (or_form φ_var (or_form φ_const (or_form φ_henkin φ_zero)))) ∈ Δ := by

change or_form A (or_form B (or_form C D)) ∈ Δ

exact h_cases_subst

have h_eq := eq_refl_in_Δ Δ h_max h_con w

have h_and := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_cases_subst') (SPA_Deduction_Internal.hyp Δ _ h_eq)

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and

· -- Caso const c

rw [encodeTerm_const]

```
    let w := codeToTerm (s := s) (Code.cons (s := s+1) 1 (Code.cons (s := s+1) (NatS_encode c) (Code.nil (s := s+1))))

    have h_closed : isClosed w = true := isClosed_codeToTerm (s := s) (Code.cons (s := s+1) 1 (Code.cons (s := s+1) (NatS_encode c) (Code.nil (s := s+1))))

    let A := substForm 1 w (substForm 0 w φ_var)

    let B := substForm 1 w (substForm 0 w φ_const)

    let C := substForm 1 w (substForm 0 w φ_henkin)

    let D := substForm 1 w (substForm 0 w φ_zero)

    change and_form (or_form A (or_form B (or_form C D))) (ArithForm.eq (substTerm 1 w w) (substTerm 1 w (ArithTerm.var 1))) ∈ Δ

    rw [substTerm_fresh w 1 w (NotOccur_of_isClosed w h_closed 1)]

    have h_cases_subst : or_form A (or_form B (or_form C D)) ∈ Δ := by

      have h_const_subst : B ∈ Δ := by

        change exists_form 2 (and_form (ArithForm.eq (substTerm 1 w w) (ArithTerm.var 2)) (ArithForm.eq w (ArithTerm.var 2))) ∈ Δ

        rw [substTerm_fresh w 1 w (NotOccur_of_isClosed w h_closed 1)]

        have h_eq_and := and_self_in_Δ Δ h_max h_con w

        have h_eq_and_ded := SPA_Deduction_Internal.hyp Δ _ h_eq_and

        let F := and_form (ArithForm.eq w (ArithTerm.var 2)) (ArithForm.eq w (ArithTerm.var 2))

        have h_free : IsFreeFor 2 w F := IsFreeFor_closed F 2 ⟨w, h_closed⟩

        have h_subst_F : substForm 2 w F = and_form (ArithForm.eq (substTerm 2 w w) w) (ArithForm.eq (substTerm 2 w w) w) := rfl

        have h_eq_and_ded' : SPA_Deduction_Internal s Δ (substForm 2 w F) := by

          rw [h_subst_F]

          rw [substTerm_fresh w 2 w (NotOccur_of_isClosed w h_closed 2)]

          exact h_eq_and_ded

        have h_exists := exists_intro Δ 2 F w h_free h_eq_and_ded'

        exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists

      have h_or3 := or_intro_left Δ B (or_form C D) (SPA_Deduction_Internal.hyp Δ _ h_const_subst)
```

```
    have h_or3_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_or3

    have h_or2 := or_intro_right Δ A (or_form B (or_form C D)) (SPA_Deduction_Internal.hyp Δ _ h_or3_in)

    exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_or2

  have h_cases_subst' : substForm 1 w (substForm 0 w (or_form φ_var (or_form φ_const (or_form φ_henkin φ_zero)))) ∈ Δ := by

    change or_form A (or_form B (or_form C D)) ∈ Δ

    exact h_cases_subst

  have h_eq := eq_refl_in_Δ Δ h_max h_con w

  have h_and := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_cases_subst') (SPA_Deduction_Internal.hyp Δ _ h_eq)

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and

· -- Caso henkin h

  rw [encodeTerm_henkin]

  let w := codeToTerm (s := s) (Code.cons (s := s+1) 2 (Code.cons (s := s+1) h.index (Code.nil (s := s+1))))

  have h_closed : isClosed w = true := isClosed_codeToTerm (s := s) (Code.cons (s := s+1) 2 (Code.cons (s := s+1) h.index (Code.nil (s := s+1))))

  let A := substForm 1 w (substForm 0 w φ_var)

  let B := substForm 1 w (substForm 0 w φ_const)

  let C := substForm 1 w (substForm 0 w φ_henkin)

  let D := substForm 1 w (substForm 0 w φ_zero)

  change and_form (or_form A (or_form B (or_form C D))) (ArithForm.eq (substTerm 1 w w) (substTerm 1 w (ArithTerm.var 1))) ∈ Δ

  rw [substTerm_fresh w 1 w (NotOccur_of_isClosed w h_closed 1)]

  have h_cases_subst : or_form A (or_form B (or_form C D)) ∈ Δ := by

    have h_henkin_subst : C ∈ Δ := by

      change exists_form 2 (and_form (ArithForm.eq (substTerm 1 w w) (ArithTerm.var 2)) (ArithForm.eq w (ArithTerm.var 2))) ∈ Δ

      rw [substTerm_fresh w 1 w (NotOccur_of_isClosed w h_closed 1)]
```

```
      have h_eq_and := and_self_in_Δ Δ h_max h_con w
      have h_eq_and_ded := SPA_Deduction_Internal.hyp Δ _ h_eq_and
      let F := and_form (ArithForm.eq w (ArithTerm.var 2)) (ArithForm.eq w (ArithTerm.var 2))
      have h_free : IsFreeFor 2 w F := IsFreeFor_closed F 2 ⟨w, h_closed⟩
      have h_subst_F : substForm 2 w F = and_form (ArithForm.eq (substTerm 2 w w) w) (ArithForm.eq (substTerm 2 w w) w) := rfl
      have h_eq_and_ded' : SPA_Deduction_Internal s Δ (substForm 2 w F) := by
        rw [h_subst_F]
        rw [substTerm_fresh w 2 w (NotOccur_of_isClosed w h_closed 2)]
        exact h_eq_and_ded
      have h_exists := exists_intro Δ 2 F w h_free h_eq_and_ded'
      exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists
    have h_or4 := or_intro_left Δ C D (SPA_Deduction_Internal.hyp Δ _ h_henkin_subst)
    have h_or4_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_or4
    have h_or3 := or_intro_right Δ B (or_form C D) (SPA_Deduction_Internal.hyp Δ _ h_or4_in)
    have h_or3_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_or3
    have h_or2 := or_intro_right Δ A (or_form B (or_form C D)) (SPA_Deduction_Internal.hyp Δ _ h_or3_in)
    exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_or2
  have h_cases_subst' : substForm 1 w (substForm 0 w (or_form φ_var (or_form φ_const (or_form φ_henkin φ_zero)))) ∈ Δ := by
    change or_form A (or_form B (or_form C D)) ∈ Δ
    exact h_cases_subst
  have h_eq := eq_refl_in_Δ Δ h_max h_con w
  have h_and := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_cases_subst') (SPA_Deduction_Internal.hyp Δ _ h_eq)
  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and
```

· -- Caso zero

rw [encodeTerm_zero]

let w := codeToTerm (s := s) (Code.cons (s := s+1) 3 (Code.nil (s := s+1)))

have h_closed : isClosed w = true := isClosed_codeToTerm (s := s) (Code.cons (s := s+1) 3 (Code.nil (s := s+1)))

let A := substForm 1 w (substForm 0 w φ_var)

let B := substForm 1 w (substForm 0 w φ_const)

let C := substForm 1 w (substForm 0 w φ_henkin)

let D := substForm 1 w (substForm 0 w φ_zero)

change and_form (or_form A (or_form B (or_form C D))) (ArithForm.eq (substTerm 1 w w) (substTerm 1 w (ArithTerm.var 1))) ∈ Δ

rw [substTerm_fresh w 1 w (NotOccur_of_isClosed w h_closed 1)]

have h_cases_subst : or_form A (or_form B (or_form C D)) ∈ Δ := by

have h_zero_subst : D ∈ Δ := by

change and_form (ArithForm.eq (substTerm 1 w w) (substTerm 1 w w)) (ArithForm.eq w (substTerm 1 w w)) ∈ Δ

rw [substTerm_fresh w 1 w (NotOccur_of_isClosed w h_closed 1)]

exact and_self_in_Δ Δ h_max h_con w

have h_or4 := or_intro_right Δ C D (SPA_Deduction_Internal.hyp Δ _ h_zero_subst)

have h_or4_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_or4

have h_or3 := or_intro_right Δ B (or_form C D) (SPA_Deduction_Internal.hyp Δ _ h_or4_in)

have h_or3_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_or3

have h_or2 := or_intro_right Δ A (or_form B (or_form C D)) (SPA_Deduction_Internal.hyp Δ _ h_or3_in)

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_or2

have h_cases_subst' : substForm 1 w (substForm 0 w (or_form φ_var (or_form φ_const (or_form φ_henkin φ_zero)))) ∈ Δ := by

change or_form A (or_form B (or_form C D)) ∈ Δ

```
      exact h_cases_subst

    have h_eq := eq_refl_in_Δ Δ h_max h_con w

    have h_and := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_cases_subst') (SPA_Deduction_Internal.hyp Δ _ h_eq)

    exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and

  · -- Condición de Funcionalidad para los casos base

   intro t c h_base h_in

   rcases c with ⟨c_val, hc⟩

   rcases h_base with ⟨n, rfl⟩ | ⟨c', rfl⟩ | ⟨h, rfl⟩ | rfl

   · -- Caso var n

    let w := codeToTerm (s := s) (Code.cons (s := s+1) 0 (Code.cons (s := s+1) n (Code.nil (s := s+1))))

    have h_closed : isClosed w = true := isClosed_codeToTerm (s := s) (Code.cons (s := s+1) 0 (Code.cons (s := s+1) n (Code.nil (s := s+1))))

    change ArithForm.eq c_val (encodeTerm (ArithTerm.var n)).val ∈ Δ

    change substForm 1 c_val (substForm 0 (encodeTerm (ArithTerm.var n)).val φ_ser_base) ∈ Δ at h_in

    rw [encodeTerm_var] at h_in ⊢

    change and_form (substForm 1 c_val (substForm 0 w (or_form φ_var (or_form φ_const (or_form φ_henkin φ_zero))))) (ArithForm.eq (substTerm 1 c_val w) (substTerm 1 c_val (ArithTerm.var 1))) ∈ Δ at h_in

    rw [substTerm_fresh (s := s) w 1 c_val (NotOccur_of_isClosed (s := s) w h_closed 1)] at h_in

    try dsimp [substTerm] at h_in

    have h_eq_in := and_elim_right Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_in)

    have h_eq_in' := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_eq_in

    apply maximal_consistent_closed_under_deduction Δ h_max h_con

    apply SPA_Deduction_Internal.eq_symm

    exact SPA_Deduction_Internal.hyp Δ _ h_eq_in'

   · -- Caso const c'
```

```
    let w := codeToTerm (s := s) (Code.cons (s := s+1) 1 (Code.cons (s := s+1) (NatS_encode c') (Code.nil (s := s+1))))

    have h_closed : isClosed w = true := isClosed_codeToTerm (s := s) (Code.cons (s := s+1) 1 (Code.cons (s := s+1) (NatS_encode c') (Code.nil (s := s+1))))

    change ArithForm.eq c_val (encodeTerm (ArithTerm.const c')).val ∈ Δ

    change substForm 1 c_val (substForm 0 (encodeTerm (ArithTerm.const c')).val φ_ser_base) ∈ Δ at h_in

    rw [encodeTerm_const] at h_in ⊢

    change and_form (substForm 1 c_val (substForm 0 w (or_form φ_var (or_form φ_const (or_form φ_henkin φ_zero))))) (ArithForm.eq (substTerm 1 c_val w) (substTerm 1 c_val (ArithTerm.var 1))) ∈ Δ at h_in

    rw [substTerm_fresh (s := s) w 1 c_val (NotOccur_of_isClosed (s := s) w h_closed 1)] at h_in

    try dsimp [substTerm] at h_in

    have h_eq_in := and_elim_right Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_in)

    have h_eq_in' := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_eq_in

    apply maximal_consistent_closed_under_deduction Δ h_max h_con

    apply SPA_Deduction_Internal.eq_symm

    exact SPA_Deduction_Internal.hyp Δ _ h_eq_in'

  · -- Caso henkin h

    let w := codeToTerm (s := s) (Code.cons (s := s+1) 2 (Code.cons (s := s+1) h.index (Code.nil (s := s+1))))

    have h_closed : isClosed w = true := isClosed_codeToTerm (s := s) (Code.cons (s := s+1) 2 (Code.cons (s := s+1) h.index (Code.nil (s := s+1))))

    change ArithForm.eq c_val (encodeTerm (ArithTerm.henkin h)).val ∈ Δ

    change substForm 1 c_val (substForm 0 (encodeTerm (ArithTerm.henkin h)).val φ_ser_base) ∈ Δ at h_in

    rw [encodeTerm_henkin] at h_in ⊢

    change and_form (substForm 1 c_val (substForm 0 w (or_form φ_var (or_form φ_const (or_form φ_henkin φ_zero))))) (ArithForm.eq (substTerm 1 c_val w) (substTerm 1 c_val (ArithTerm.var 1))) ∈ Δ at h_in
```

rw [substTerm_fresh (s := s) w 1 c_val (NotOccur_of_isClosed (s := s) w h_closed 1)] at h_in

try dsimp [substTerm] at h_in

have h_eq_in := and_elim_right Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_in)

have h_eq_in' := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_eq_in

apply maximal_consistent_closed_under_deduction Δ h_max h_con

apply SPA_Deduction_Internal.eq_symm

exact SPA_Deduction_Internal.hyp Δ _ h_eq_in'

· -- Caso zero

let w := codeToTerm (s := s) (Code.cons (s := s+1) 3 (Code.nil (s := s+1)))

have h_closed : isClosed w = true := isClosed_codeToTerm (s := s) (Code.cons (s := s+1) 3 (Code.nil (s := s+1)))

change ArithForm.eq c_val (encodeTerm ArithTerm.zero).val ∈ Δ

change substForm 1 c_val (substForm 0 (encodeTerm ArithTerm.zero).val φ_ser_base) ∈ Δ at h_in

rw [encodeTerm_zero] at h_in ⊢

change and_form (substForm 1 c_val (substForm 0 w (or_form φ_var (or_form φ_const (or_form φ_henkin φ_zero))))) (ArithForm.eq (substTerm 1 c_val w) (substTerm 1 c_val (ArithTerm.var 1))) ∈ Δ at h_in

rw [substTerm_fresh (s := s) w 1 c_val (NotOccur_of_isClosed (s := s) w h_closed 1)] at h_in

try dsimp [substTerm] at h_in

have h_eq_in := and_elim_right Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_in)

have h_eq_in' := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_eq_in

apply maximal_consistent_closed_under_deduction Δ h_max h_con

apply SPA_Deduction_Internal.eq_symm

exact SPA_Deduction_Internal.hyp Δ _ h_eq_in'

Lema `serializeTerm_representable`: Representabilidad de la serialización mediante igualdad simple

Se demuestra que para cualquier término t del nivel $s+1$, la función que mapea t a su código interno es representable en Δ mediante la fórmula más simple posible: la igualdad $v_0 = v_1$. Formalmente:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal}(\Delta) \land \text{HenkinSet}(\Delta) \land \text{Consistent}(\Delta) \implies \forall t \in \text{ArithTerm}(s+1), $$

existe una fórmula φ (concretamente $v_0 = v_1$) que satisface las condiciones de `IsRepresentable` para la función $f(t) = \text{internalCode}(\text{toSPACode}(\text{serializeTerm}(t)))$ de manera puntual, es decir, para el término t específico.

Estructura de la demostración

La prueba construye $\varphi := (v_0 = v_1)$ y verifica las cinco condiciones requeridas:

1. La variable v_0 es libre en φ: Se demuestra por contradicción. Si v_0 no fuera libre en $v_0 = v_1$, la definición de `NotFreeForm` exigiría que v_0 no ocurriera en v_0 (como término izquierdo de la igualdad), lo cual es falso pues v_0 es exactamente el término `var 0`. La regla `NotOccur.var 0 h_ne` requiere $0 \neq 0$, lo cual es una contradicción inmediata.

2. La variable v_1 es libre en φ: Análogo al caso anterior: v_1 aparece como término derecho de la igualdad, por lo que no puede ser no libre.

3. Ninguna otra variable es libre: Para cualquier $v \neq 0, 1$, se tiene $\text{NotOccur.var } 0\ hv0$ (pues $v \neq 0$) y $\text{NotOccur.var } 1\ hv1$ (pues $v \neq 1$). Esto es suficiente para establecer $\text{NotFreeForm } v\ (v_0 = v_1)$.

---

4. Condición de gráfico: Se debe demostrar que la instancia $\varphi[v_0 := \text{encodeTerm}(t),\ v_1 := \text{encodeTerm}(t)]$ pertenece a Δ. La sustitución produce:

$$ \text{encodeTerm}(t) = \text{encodeTerm}(t) $$

Para reescribir correctamente, se utiliza el lema `substTerm_fresh`: dado que $\text{encodeTerm}(t)$ es un término cerrado (por ser un `TermClosed`), la variable 1 no ocurre en él. Por tanto, la sustitución de v_1 por $\text{encodeTerm}(t)$ en el lado derecho de la igualdad (que es v_1) simplemente devuelve $\text{encodeTerm}(t)$. La fórmula resultante es la igualdad reflexiva, que pertenece a Δ por el lema `eq_refl_in_Δ`.

---

5. Condición de funcionalidad: Para cualquier término cerrado c tal que $\varphi[v_0 := \text{encodeTerm}(t),\ v_1 := c] \in \Delta$, se debe demostrar $c = \text{encodeTerm}(t) \in \Delta$. La instancia de φ es:

$$ \text{encodeTerm}(t) = c $$

Nuevamente, `substTerm_fresh` garantiza que la sustitución de v_1 por c en el lado derecho produce c sin alterar el lado izquierdo. Por hipótesis, esta igualdad

pertenece a Δ. Aplicando simetría de la igualdad (`eq_symm`), se obtiene $c = \text{encodeTerm}(t)$, que también pertenece a Δ por clausura deductiva (maximalidad y consistencia).

Intuición y necesidad

Este lema revela una propiedad fundamental de la codificación en sistemas formales: cuando la función a representar asigna a cada objeto un código único e inyectivo, la relación "y es el código de x" puede representarse simplemente como la igualdad $x = y$, siempre que trabajemos internamente con los códigos mismos. Es decir, en lugar de construir una fórmula compleja que describa el proceso de serialización, podemos usar la identidad porque el argumento de entrada ya es el código.

La aparente trivialidad de la fórmula $\varphi = (v_0 = v_1)$ es engañosa: su poder reside en que `encodeTerm(t)` ya realiza la codificación en el metalenguaje. Cuando instanciamos v_0 con $\text{encodeTerm}(t)$, estamos insertando directamente el numeral que codifica t. La condición de gráfico se satisface porque la igualdad reflexiva es un teorema; la condición de funcionalidad se satisface porque si c iguala al código de t, entonces c es el código de t. Esta construcción es un caso particular de un principio más general: para cualquier función inyectiva f, la relación $y = f(x)$ es representable mediante $v_0 = v_1$ cuando v_0 se instancia con $f(x)$.

Este resultado contrasta con el lema `serializeTerm_base_representable`, donde la fórmula `φ_ser_base` describía explícitamente la estructura de los códigos base mediante disyunciones. La diferencia radica en el nivel de abstracción: `φ_ser_base` funciona para cualquier par de códigos y determina si uno corresponde a un caso base, mientras que aquí la fórmula solo funciona para el código específico de t.

```
lemma serializeTerm_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal
Δ)
   (_h_henkin : HenkinSet Δ) (h_con : Consistent Δ) (t : ArithTerm (s+1)) :
   ∃ (φ : ArithForm s)
    (_h_v0 : ¬ NotFreeForm 0 φ)
    (_h_v1 : ¬ NotFreeForm 1 φ)
    (_h_only : ∀ (v : Nat), v ≠ 0 → v ≠ 1 → NotFreeForm v φ),
    -- Gráfico: φ(encodeTerm t, internalCode (toSPACode (serializeTerm t))) ∈ Δ
    (substForm 1 (encodeTerm t).val (substForm 0 (encodeTerm t).val φ) ∈ Δ) ∧
    -- Funcionalidad: si φ(encodeTerm t, c) ∈ Δ, entonces c = internalCode (toSPACode
(serializeTerm t))
    (∀ (c : TermClosed s),
     substForm 1 c.val (substForm 0 (encodeTerm t).val φ) ∈ Δ →
     ArithForm.eq c.val (encodeTerm t).val ∈ Δ) := by
 let φ : ArithForm s := ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 1)
 use φ
 have h_v0 : ¬ NotFreeForm 0 φ := by
  intro h
  dsimp [φ, NotFreeForm] at h
  rcases h with ⟨h0, _⟩
  cases h0 with
  | var m h_ne => exact h_ne rfl
 have h_v1 : ¬ NotFreeForm 1 φ := by
  intro h
  dsimp [φ, NotFreeForm] at h
  rcases h with ⟨_, h1⟩
  cases h1 with
  | var m h_ne => exact h_ne rfl
 have h_only : ∀ (v : Nat), v ≠ 0 → v ≠ 1 → NotFreeForm v φ := by
```

```
  intro v hv0 hv1
  dsimp [φ, NotFreeForm]
  constructor
  · exact NotOccur.var 0 hv0
  · exact NotOccur.var 1 hv1
 refine ⟨h_v0, h_v1, h_only, ⟨?_, ?_⟩⟩
 · have h_eq_subst : substForm 1 (encodeTerm t).val (substForm 0 (encodeTerm t).val φ) = ArithForm.eq (encodeTerm t).val (encodeTerm t).val := by
    dsimp [φ, substForm, substTerm]
    have h_not_occur : NotOccur 1 (encodeTerm t).val := NotOccur_of_isClosed (encodeTerm t).val (encodeTerm t).property 1
    rw [substTerm_fresh (encodeTerm t).val 1 (encodeTerm t).val h_not_occur]
  rw [h_eq_subst]
  exact eq_refl_in_Δ Δ h_max h_con (encodeTerm t).val
 · intro c h_in
  have h_eq_subst2 (c : TermClosed s) : substForm 1 c.val (substForm 0 (encodeTerm t).val φ) = ArithForm.eq (encodeTerm t).val c.val := by
    dsimp [φ, substForm, substTerm]
    have h_not_occur : NotOccur 1 (encodeTerm t).val := NotOccur_of_isClosed (encodeTerm t).val (encodeTerm t).property 1
    rw [substTerm_fresh (encodeTerm t).val 1 c.val h_not_occur]
  have h_ded : SPA_Deduction_Internal s Δ (ArithForm.eq (encodeTerm t).val c.val) :=
    SPA_Deduction_Internal.hyp Δ _ (by rwa [h_eq_subst2 c] at h_in)
  have h_symm := SPA_Deduction_Internal.eq_symm Δ (encodeTerm t).val c.val h_ded
  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_symm
```

Lema `serializeForm_representable`: Representabilidad de la serialización de fórmulas mediante igualdad simple

Se demuestra el análogo del lema anterior para fórmulas: para cualquier fórmula φ del nivel $s+1$, la función que mapea φ a su código interno es representable en Δ mediante la fórmula de igualdad $v_0 = v_1$. Formalmente:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal}(\Delta) \land \text{HenkinSet}(\Delta) \land \text{Consistent}(\Delta) \implies \forall \varphi \in \text{ArithForm}(s+1), $$

existe una fórmula ψ (concretamente $v_0 = v_1$) que satisface las condiciones de representabilidad puntual para la función $g(\varphi) = \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\varphi)))$, es decir, la codificación de Gödel para fórmulas.

Estructura de la demostración

La prueba es estructuralmente idéntica a la de `serializeTerm_representable`, reemplazando sistemáticamente `encodeTerm` por `encodeForm` y adaptando los nombres de variables. Se construye $\psi := (v_0 = v_1)$ y se verifican las cinco condiciones:

1. Variable v_0 libre: La igualdad $v_0 = v_1$ contiene a v_0 como término izquierdo. Si v_0 no fuera libre, la definición de `NotFreeForm` exigiría $\text{NotOccur } 0\ (\text{var } 0)$, lo cual es falso por la regla `NotOccur.var` que requiere índices distintos.

2. Variable v_1 libre: Análogo, utilizando que v_1 aparece como término derecho de la igualdad.

3. Exclusividad de variables libres: Para cualquier $v \neq 0, 1$, tanto v_0 como v_1 son términos que no contienen a v (pues son variables de índice distinto), por lo que $\text{NotFreeForm } v\ (v_0 = v_1)$ se satisface.

---

4. Condición de gráfico: Se debe demostrar $\psi[v_0 := \text{encodeForm}(\varphi),\ v_1 := \text{encodeForm}(\varphi)] \in \Delta$. La sustitución produce:

$$ \text{encodeForm}(\varphi) = \text{encodeForm}(\varphi) $$

La reescritura utiliza `substTerm_fresh` con el hecho de que $\text{encodeForm}(\varphi)$ es un `TermClosed` y por tanto la variable 1 no ocurre en él. El resultado es una igualdad reflexiva que pertenece a Δ por `eq_refl_in_Δ`.

---

5. Condición de funcionalidad: Para cualquier término cerrado c tal que $\psi[v_0 := \text{encodeForm}(\varphi),\ v_1 := c] \in \Delta$, se debe demostrar $c = \text{encodeForm}(\varphi) \in \Delta$. La instancia es:

$$ \text{encodeForm}(\varphi) = c $$

Por hipótesis, esta igualdad pertenece a Δ. Aplicando simetría, se obtiene $c = \text{encodeForm}(\varphi) \in \Delta$.

Intuición y necesidad

Este lema completa el par de resultados de representabilidad puntual mediante igualdad, cubriendo tanto términos como fórmulas. La simetría entre `serializeTerm_representable` y `serializeForm_representable` no es accidental: ambos se basan en el mismo principio de que la codificación interna —al ser una inyección del lenguaje objeto en los numerales del nivel inferior— permite representar la relación "y es el código de x" como la identidad $x = y$ cuando trabajamos con los códigos ya computados.

La importancia de este resultado para el desarrollo global de la Fase 1C radica en que proporciona una fórmula representadora mínima y universal. Mientras que `serializeTerm_base_representable` construía una fórmula compleja con disyunciones para capturar la estructura de los casos base, estos lemas muestran que para un objeto sintáctico fijo, la igualdad simple es suficiente. Esta observación es crucial para los argumentos de diagonalización posteriores, donde se necesitará representar la función que asigna a cada fórmula su código de Gödel de la manera más simple posible, minimizando la complejidad de la fórmula diagonal resultante. La clausura deductiva de Δ (maximalidad y consistencia) garantiza que las igualdades verdaderas entre códigos son demostrables y que la funcionalidad se preserva.

lemma serializeForm_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)

(_h_henkin : HenkinSet Δ) (h_con : Consistent Δ) (φ : ArithForm (s+1)) :

∃ (ψ : ArithForm s)

(_h_v0 : ¬ NotFreeForm 0 ψ)

(_h_v1 : ¬ NotFreeForm 1 ψ)

(_h_only : ∀ (v : Nat), v ≠ 0 → v ≠ 1 → NotFreeForm v ψ),

(substForm 1 (encodeForm φ).val (substForm 0 (encodeForm φ).val ψ) ∈ Δ) ∧

(∀ (c : TermClosed s),

```
      substForm 1 c.val (substForm 0 (encodeForm φ).val ψ) ∈ Δ →
      ArithForm.eq c.val (encodeForm φ).val ∈ Δ) := by
  let ψ_eq : ArithForm s := ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 1)
  use ψ_eq
  have h_v0 : ¬ NotFreeForm 0 ψ_eq := by
    intro h
    dsimp [ψ_eq, NotFreeForm] at h
    rcases h with ⟨h0, _⟩
    cases h0 with
    | var m h_ne => exact h_ne rfl
  have h_v1 : ¬ NotFreeForm 1 ψ_eq := by
    intro h
    dsimp [ψ_eq, NotFreeForm] at h
    rcases h with ⟨_, h1⟩
    cases h1 with
    | var m h_ne => exact h_ne rfl
  have h_only : ∀ (v : Nat), v ≠ 0 → v ≠ 1 → NotFreeForm v ψ_eq := by
    intro v hv0 hv1
    dsimp [ψ_eq, NotFreeForm]
    constructor
    · exact NotOccur.var 0 hv0
    · exact NotOccur.var 1 hv1
  refine ⟨h_v0, h_v1, h_only, ⟨?_, ?_⟩⟩
  · have h_eq_subst : substForm 1 (encodeForm φ).val (substForm 0 (encodeForm
φ).val ψ_eq) = ArithForm.eq (encodeForm φ).val (encodeForm φ).val := by
      dsimp [ψ_eq, substForm, substTerm]
      have h_not_occur : NotOccur 1 (encodeForm φ).val := NotOccur_of_isClosed
(encodeForm φ).val (encodeForm φ).property 1
      rw [substTerm_fresh (encodeForm φ).val 1 (encodeForm φ).val h_not_occur]
```

rw [h_eq_subst]

exact eq_refl_in_Δ Δ h_max h_con (encodeForm φ).val

· intro c h_in

have h_eq_subst2 (c : TermClosed s) : substForm 1 c.val (substForm 0 (encodeForm φ).val ψ_eq) = ArithForm.eq (encodeForm φ).val c.val := by

dsimp [ψ_eq, substForm, substTerm]

have h_not_occur : NotOccur 1 (encodeForm φ).val := NotOccur_of_isClosed (encodeForm φ).val (encodeForm φ).property 1

rw [substTerm_fresh (encodeForm φ).val 1 c.val h_not_occur]

have h_ded : SPA_Deduction_Internal s Δ (ArithForm.eq (encodeForm φ).val c.val) :=

SPA_Deduction_Internal.hyp Δ _ (by rwa [h_eq_subst2 c] at h_in)

have h_symm := SPA_Deduction_Internal.eq_symm Δ (encodeForm φ).val c.val h_ded

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_symm

Lema `substTerm_representable`: Representabilidad de la sustitución de términos

Se demuestra que la operación de sustitución sintáctica de una variable por un término es representable en Δ. Específicamente, para una variable v y términos u, t del nivel $s+1$, la función que mapea el código de t al código del resultado de sustituir v por u en t es representable:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } } s,\ \text{Maximal}(\Delta) \land \text{HenkinSet}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \forall v \in \mathbb{N},\ \forall u, t \in \text{ArithTerm}(s+1),\ \exists \varphi \in \text{ArithForm } } s, $$

tal que φ satisface las condiciones de representabilidad puntual para la función $t \mapsto \text{substTerm}(v, u, t)$ en términos de sus códigos: $\text{encodeTerm}(t) \mapsto \text{encodeTerm}(\text{substTerm}(v, u, t))$.

La fórmula representadora

A diferencia de los lemas anteriores que utilizaban la igualdad simple, aquí se requiere una fórmula más elaborada:

$$ \varphi := (\text{encodeTerm}(t) = v_0) \to (\text{encodeTerm}(\text{substTerm}(v, u, t)) = v_1) $$

o, en la notación del lenguaje objeto:

$$ \varphi := (v_0 = \overline{\text{encodeTerm}(t)}) \to (v_1 = \overline{\text{encodeTerm}(\text{substTerm}(v, u, t))}) $$

Intuición y necesidad

Esta construcción captura la esencia de la representabilidad de funciones en teorías aritméticas: para representar una función $f(x) = y$, se utiliza una implicación donde el antecedente identifica la entrada x y el consecuente identifica la salida y. La fórmula φ expresa: "si v_0 es el código de t, entonces v_1 es el código de $\text{substTerm}(v, u, t)$". Los códigos concretos de t y de $\text{substTerm}(v, u, t)$ están incrustados directamente en la fórmula como numerales (términos cerrados), lo cual es posible porque el lema es puntual: se construye una fórmula específica para cada triple (v, u, t).

Estructura de la demostración

1. Variable v_0 libre: La variable v_0 aparece en el antecedente de la implicación, concretamente en la igualdad $v_0 = \overline{\text{encodeTerm}(t)}$. Si v_0 no fuera libre, la definición de `NotFreeForm` exigiría que v_0 no ocurriera en el término izquierdo de esta igualdad, lo cual es falso.

2. Variable v_1 libre: Análogo, utilizando que v_1 aparece en el consecuente de la implicación.

3. Exclusividad de variables libres: Para cualquier $v' \neq 0, 1$, se debe demostrar que v' no ocurre libre en φ. La implicación requiere que v' no sea libre en ninguna de las dos igualdades. Cada igualdad contiene una variable (v_0 o v_1) y un numeral cerrado (`encodeTerm(t).val` o `encodeTerm(substTerm v u t).val`). Dado que $v' \neq 0, 1$, no aparece en las variables; y dado que los numerales son términos cerrados (por ser `TermClosed`), v' no ocurre en ellos por el lema `NotOccur_of_isClosed`. La conjunción de estas condiciones establece `NotFreeForm v' φ`.

---

4. Condición de gráfico: Se debe demostrar $\varphi[v_0 := \text{encodeTerm}(t),\ v_1 := \text{encodeTerm}(\text{substTerm}(v, u, t))] \in \Delta$. La sustitución produce:

$$ (\text{encodeTerm}(t) = \text{encodeTerm}(t)) \to (\text{encodeTerm}(\text{substTerm}(v, u, t)) = \text{encodeTerm}(\text{substTerm}(v, u, t))) $$

Ambos lados de la implicación son igualdades reflexivas. La reescritura utiliza `substTerm_fresh` cuatro veces para verificar que las sustituciones no alteran los numerales cerrados (pues ninguna variable de sustitución ocurre en ellos). El consecuente de la implicación pertenece a Δ por `eq_refl_in_Δ`. Mediante introducción de la implicación (`imp_intro`), se obtiene que la implicación completa es demostrable y por tanto pertenece a Δ por clausura deductiva.

---

5. Condición de funcionalidad: Para cualquier término cerrado c tal que $\varphi[v_0 := \text{encodeTerm}(t),\ v_1 := c] \in \Delta$, se debe demostrar $c = \text{encodeTerm}(\text{substTerm}(v, u, t)) \in \Delta$. La instancia de φ es:

$$ (\text{encodeTerm}(t) = \text{encodeTerm}(t)) \to (c = \text{encodeTerm}(\text{substTerm}(v, u, t))) $$

Por hipótesis, esta implicación pertenece a Δ. El antecedente $(\text{encodeTerm}(t) = \text{encodeTerm}(t))$ también pertenece a Δ (por reflexividad). Aplicando eliminación de la implicación (`imp_elim`), se deduce que el consecuente $c = \text{encodeTerm}(\text{substTerm}(v, u, t))$ pertenece a Δ.

Significado para el programa de Gödelización

Este lema es un paso fundamental hacia la representabilidad de las operaciones sintácticas básicas. En la jerarquía de la aritmetización de la sintaxis, la sustitución es una de las operaciones más importantes, pues permite expresar la autorreferencia: la fórmula diagonal $\psi(\overline{\psi})$ se construye precisamente sustituyendo una variable por el código de una fórmula. Al demostrar que la sustitución es representable, se establece que el sistema formal puede "hablar internamente" sobre el proceso de sustitución, lo cual es indispensable para construir el análogo estratificado del lema de diagonalización de Gödel-Carnap. La estructura de la fórmula representadora como implicación entre igualdades refleja el carácter condicional de la representación de funciones: la salida está determinada unívocamente por la entrada, y la fórmula captura exactamente esta dependencia funcional.

```
lemma substTerm_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
   (_h_henkin : HenkinSet Δ) (h_con : Consistent Δ) (v : Nat) (u t : ArithTerm (s+1)) :
   ∃ (φ : ArithForm s)
    (_h_v0 : ¬ NotFreeForm 0 φ)
    (_h_v1 : ¬ NotFreeForm 1 φ)
    (_h_only : ∀ (v' : Nat), v' ≠ 0 → v' ≠ 1 → NotFreeForm v' φ),
    (substForm 1 (encodeTerm (substTerm v u t)).val (substForm 0 (encodeTerm t).val φ) ∈ Δ) ∧
    (∀ (c : TermClosed s),
     substForm 1 c.val (substForm 0 (encodeTerm t).val φ) ∈ Δ →
     ArithForm.eq c.val (encodeTerm (substTerm v u t)).val ∈ Δ) := by
 let φ : ArithForm s := ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (encodeTerm t).val)
                        (ArithForm.eq (ArithTerm.var 1) (encodeTerm (substTerm v u t)).val)
 use φ
 have h_v0 : ¬ NotFreeForm 0 φ := by
  intro h
  dsimp [φ, NotFreeForm] at h
  rcases h with ⟨h0, _⟩
  rcases h0 with ⟨h00, _⟩
  cases h00 with
  | var m h_ne => exact h_ne rfl
 have h_v1 : ¬ NotFreeForm 1 φ := by
  intro h
  dsimp [φ, NotFreeForm] at h
```

```
  rcases h with ⟨_, h1⟩
  rcases h1 with ⟨h11, _⟩
  cases h11 with
  | var m h_ne => exact h_ne rfl
 have h_only : ∀ (v' : Nat), v' ≠ 0 → v' ≠ 1 → NotFreeForm v' φ := by
  intro v' hv'0 hv'1
  dsimp [φ, NotFreeForm]
  refine ⟨⟨NotOccur.var 0 hv'0, ?_⟩, ⟨NotOccur.var 1 hv'1, ?_⟩⟩
  · exact NotOccur_of_isClosed _ (encodeTerm t).property v'
  · exact NotOccur_of_isClosed _ (encodeTerm (substTerm v u t)).property v'
 refine ⟨h_v0, h_v1, h_only, ⟨?_, ?_⟩⟩
 · have h_eq_subst : substForm 1 (encodeTerm (substTerm v u t)).val (substForm 0 (encodeTerm t).val φ) =
            ArithForm.impl (ArithForm.eq (encodeTerm t).val (encodeTerm t).val)
                (ArithForm.eq (encodeTerm (substTerm v u t)).val (encodeTerm (substTerm v u t)).val) := by
    dsimp [φ, substForm, substTerm]
    have h0 : NotOccur 0 (encodeTerm t).val := NotOccur_of_isClosed _ (encodeTerm t).property 0
    have h1 : NotOccur 0 (encodeTerm (substTerm v u t)).val := NotOccur_of_isClosed _ (encodeTerm (substTerm v u t)).property 0
    have h2 : NotOccur 1 (encodeTerm t).val := NotOccur_of_isClosed _ (encodeTerm t).property 1
    have h3 : NotOccur 1 (encodeTerm (substTerm v u t)).val := NotOccur_of_isClosed _ (encodeTerm (substTerm v u t)).property 1
    rw [substTerm_fresh (encodeTerm t).val 0 (encodeTerm t).val h0]
    rw [substTerm_fresh (encodeTerm (substTerm v u t)).val 0 (encodeTerm t).val h1]
    rw [substTerm_fresh (encodeTerm t).val 1 (encodeTerm (substTerm v u t)).val h2]
    rw [substTerm_fresh (encodeTerm (substTerm v u t)).val 1 (encodeTerm (substTerm v u t)).val h3]
  rw [h_eq_subst]
```

```
  have h_refl1 : ArithForm.eq (encodeTerm (substTerm v u t)).val (encodeTerm (substTerm v u t)).val ∈ Δ :=

    eq_refl_in_Δ Δ h_max h_con (encodeTerm (substTerm v u t)).val

  have h_ded : SPA_Deduction_Internal s (insert (ArithForm.eq (encodeTerm t).val (encodeTerm t).val) Δ) (ArithForm.eq (encodeTerm (substTerm v u t)).val (encodeTerm (substTerm v u t)).val) := by

    apply SPA_Deduction_Internal.weaken Δ

    · exact SPA_Deduction_Internal.hyp Δ _ h_refl1

    · exact Set.subset_insert _ _

  have h_imp_ded : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq (encodeTerm t).val (encodeTerm t).val) (ArithForm.eq (encodeTerm (substTerm v u t)).val (encodeTerm (substTerm v u t)).val)) :=

    SPA_Deduction_Internal.imp_intro Δ _ _ h_ded

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_imp_ded

 · intro c h_in

  have h_eq_subst2 (c : TermClosed s) : substForm 1 c.val (substForm 0 (encodeTerm t).val φ) =

                                ArithForm.impl (ArithForm.eq (encodeTerm t).val (encodeTerm t).val)

                                        (ArithForm.eq c.val (encodeTerm (substTerm v u t)).val) := by

    dsimp [φ, substForm, substTerm]

    have h0 : NotOccur 0 (encodeTerm t).val := NotOccur_of_isClosed _ (encodeTerm t).property 0

    have h1 : NotOccur 0 (encodeTerm (substTerm v u t)).val := NotOccur_of_isClosed _ (encodeTerm (substTerm v u t)).property 0

    have h2 : NotOccur 1 (encodeTerm t).val := NotOccur_of_isClosed _ (encodeTerm t).property 1

    have h3 : NotOccur 1 (encodeTerm (substTerm v u t)).val := NotOccur_of_isClosed _ (encodeTerm (substTerm v u t)).property 1

    rw [substTerm_fresh (encodeTerm t).val 0 (encodeTerm t).val h0]

    rw [substTerm_fresh (encodeTerm (substTerm v u t)).val 0 (encodeTerm t).val h1]

    rw [substTerm_fresh (encodeTerm t).val 1 c.val h2]
```

```
    rw [substTerm_fresh (encodeTerm (substTerm v u t)).val 1 c.val h3]

  have h_impl_in : ArithForm.impl (ArithForm.eq (encodeTerm t).val (encodeTerm t).val) (ArithForm.eq c.val (encodeTerm (substTerm v u t)).val) ∈ Δ := by

    rwa [h_eq_subst2 c] at h_in

  have h_impl_ded : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq (encodeTerm t).val (encodeTerm t).val) (ArithForm.eq c.val (encodeTerm (substTerm v u t)).val)) :=

    SPA_Deduction_Internal.hyp Δ _ h_impl_in

  have h_refl0_ded : SPA_Deduction_Internal s Δ (ArithForm.eq (encodeTerm t).val (encodeTerm t).val) :=

    SPA_Deduction_Internal.hyp Δ _ (eq_refl_in_Δ Δ h_max h_con (encodeTerm t).val)

  have h_ded : SPA_Deduction_Internal s Δ (ArithForm.eq c.val (encodeTerm (substTerm v u t)).val) :=

    SPA_Deduction_Internal.imp_elim Δ _ _ h_impl_ded h_refl0_ded

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_ded
```

Lema `substForm_representable`: Representabilidad de la sustitución de fórmulas

Se demuestra el análogo del lema anterior para el nivel de las fórmulas: la operación de sustitución sintáctica de una variable por un término dentro de una fórmula es representable en Δ. Formalmente:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal}(\Delta) \land \text{HenkinSet}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \forall v \in \mathbb{N},\ \forall u \in \text{ArithTerm}(s+1),\ \forall \varphi \in \text{ArithForm}(s+1),\ \exists \psi \in \text{ArithForm } } s, $$

tal que ψ satisface las condiciones de representabilidad puntual para la función $\varphi \mapsto \text{substForm}(v, u, \varphi)$ en términos de sus códigos: $\text{encodeForm}(\varphi) \mapsto \text{encodeForm}(\text{substForm}(v, u, \varphi))$.

La fórmula representadora

La construcción es estructuralmente idéntica a la de `substTerm_representable`, adaptada al tipo `ArithForm`:

$$ \psi := (\text{encodeForm}(\varphi) = v_0) \to (\text{encodeForm}(\text{substForm}(v, u, \varphi)) = v_1) $$

es decir:

$$ \psi := (v_0 = \overline{\text{encodeForm}(\varphi)}) \to (v_1 = \overline{\text{encodeForm}(\text{substForm}(v, u, \varphi))}) $$

Estructura de la demostración

La prueba sigue el mismo esquema que `substTerm_representable`, reemplazando sistemáticamente `encodeTerm` por `encodeForm` y `ArithTerm` por `ArithForm` donde corresponda:

1. Variable v_0 libre: La variable v_0 aparece en el término izquierdo de la igualdad antecedente $(v_0 = \overline{\text{encodeForm}(\varphi)})$. La hipótesis de que v_0 no es libre fuerza $\text{NotOccur } 0\ (\text{var } 0)$, lo cual es contradictorio.

2. Variable v_1 libre: Análogo, utilizando la igualdad consecuente $(v_1 = \overline{\text{encodeForm}(\text{substForm}(v, u, \varphi))})$.

3. Exclusividad de variables libres: Para cualquier $v' \neq 0, 1$, se debe demostrar que v' no es libre en ψ. Cada igualdad contiene una variable (v_0 o v_1) y un numeral cerrado. La variable no es v' por hipótesis; el numeral es cerrado (por ser el valor subyacente de un `TermClosed`), por lo que v' no ocurre en él. La implicación preserva la no libertad, estableciendo $\text{NotFreeForm } v'\ \psi$.

---

4. Condición de gráfico: La instancia $\psi[v_0 := \text{encodeForm}(\varphi),\ v_1 := \text{encodeForm}(\text{substForm}(v, u, \varphi))]$ produce:

$$ (\text{encodeForm}(\varphi) = \text{encodeForm}(\varphi)) \to (\text{encodeForm}(\text{substForm}(v, u, \varphi)) = \text{encodeForm}(\text{substForm}(v, u, \varphi))) $$

La reescritura requiere cuatro aplicaciones de `substTerm_fresh`, justificadas por el hecho de que las variables 0 y 1 no ocurren en los numerales cerrados (por ser estos términos sin variables libres). El consecuente es una igualdad reflexiva que pertenece a Δ por `eq_refl_in_Δ`. Mediante introducción de la implicación, la fórmula completa también pertenece a Δ.

---

5. Condición de funcionalidad: Para cualquier término cerrado c con $\psi[v_0 := \text{encodeForm}(\varphi),\ v_1 := c] \in \Delta$, se instancia:

$$ (\text{encodeForm}(\varphi) = \text{encodeForm}(\varphi)) \to (c = \text{encodeForm}(\text{substForm}(v, u, \varphi))) $$

Tanto la implicación como el antecedente reflexivo pertenecen a Δ. Por eliminación de la implicación, se obtiene $c = \text{encodeForm}(\text{substForm}(v, u, \varphi)) \in \Delta$.

Significado para la aritmetización de la sintaxis

Este lema, junto con `substTerm_representable`, completa la demostración de que las dos operaciones sintácticas fundamentales —sustitución en términos y sustitución en fórmulas— son representables en cualquier teoría maximalmente consistente con propiedad de Henkin. La representabilidad de la sustitución es el pilar técnico sobre el que se erige el lema de diagonalización: para construir una fórmula ψ que "hable de sí misma", se necesita expresar internamente la operación que toma el código de una fórmula φ y produce el código de $\varphi[v := \overline{\text{encodeForm}(\varphi)}]$. Los lemas aquí demostrados establecen que esta operación, aunque compleja desde el punto de vista sintáctico, es capturable mediante una fórmula del lenguaje objeto con exactamente dos variables libres.

La estructura uniforme de la fórmula representadora como una implicación entre igualdades refleja un principio general de la teoría de la representabilidad: las funciones computables (y la sustitución sintáctica lo es) pueden representarse mediante fórmulas que expresan condicionalmente la relación entrada-salida, utilizando los códigos de los objetos concretos como parámetros numerales dentro de la fórmula. La maximalidad de Δ garantiza que estas fórmulas, una vez construidas, son efectivamente funcionales dentro de la teoría.

```
lemma substForm_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
  (_h_henkin : HenkinSet Δ) (h_con : Consistent Δ) (v : Nat) (u : ArithTerm (s+1)) (φ : ArithForm (s+1)) :
  ∃ (ψ : ArithForm s)
    (_h_v0 : ¬ NotFreeForm 0 ψ)
    (_h_v1 : ¬ NotFreeForm 1 ψ)
    (_h_only : ∀ (v' : Nat), v' ≠ 0 → v' ≠ 1 → NotFreeForm v' ψ),
    (substForm 1 (encodeForm (substForm v u φ)).val (substForm 0 (encodeForm φ).val ψ) ∈ Δ) ∧
    (∀ (c : TermClosed s),
      substForm 1 c.val (substForm 0 (encodeForm φ).val ψ) ∈ Δ →
      ArithForm.eq c.val (encodeForm (substForm v u φ)).val ∈ Δ) := by
 let ψ : ArithForm s := ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (encodeForm φ).val)
                          (ArithForm.eq (ArithTerm.var 1) (encodeForm (substForm v u φ)).val)
 use ψ
 have h_v0 : ¬ NotFreeForm 0 ψ := by
  intro h
  dsimp [ψ, NotFreeForm] at h
  rcases h with ⟨h0, _⟩
  rcases h0 with ⟨h00, _⟩
  cases h00 with
  | var m h_ne => exact h_ne rfl
 have h_v1 : ¬ NotFreeForm 1 ψ := by
  intro h
  dsimp [ψ, NotFreeForm] at h
  rcases h with ⟨_, h1⟩
  rcases h1 with ⟨h11, _⟩
  cases h11 with
```

```
  | var m h_ne => exact h_ne rfl
 have h_only : ∀ (v' : Nat), v' ≠ 0 → v' ≠ 1 → NotFreeForm v' ψ := by
  intro v' hv'0 hv'1
  dsimp [ψ, NotFreeForm]
  refine ⟨⟨NotOccur.var 0 hv'0, ?_⟩, ⟨NotOccur.var 1 hv'1, ?_⟩⟩
  · exact NotOccur_of_isClosed _ (encodeForm φ).property v'
  · exact NotOccur_of_isClosed _ (encodeForm (substForm v u φ)).property v'
 refine ⟨h_v0, h_v1, h_only, ⟨?_, ?_⟩⟩
 · have h_eq_subst : substForm 1 (encodeForm (substForm v u φ)).val (substForm 0
(encodeForm φ).val ψ) =
            ArithForm.impl (ArithForm.eq (encodeForm φ).val (encodeForm φ).val)
                (ArithForm.eq (encodeForm (substForm v u φ)).val (encodeForm
(substForm v u φ)).val) := by
    dsimp [ψ, substForm, substTerm]
    have h0 : NotOccur 0 (encodeForm φ).val := NotOccur_of_isClosed _ (encodeForm
φ).property 0
    have h1 : NotOccur 0 (encodeForm (substForm v u φ)).val := NotOccur_of_isClosed
_ (encodeForm (substForm v u φ)).property 0
    have h2 : NotOccur 1 (encodeForm φ).val := NotOccur_of_isClosed _ (encodeForm
φ).property 1
    have h3 : NotOccur 1 (encodeForm (substForm v u φ)).val := NotOccur_of_isClosed
_ (encodeForm (substForm v u φ)).property 1
    rw [substTerm_fresh (encodeForm φ).val 0 (encodeForm φ).val h0]
    rw [substTerm_fresh (encodeForm (substForm v u φ)).val 0 (encodeForm φ).val
h1]
    rw [substTerm_fresh (encodeForm φ).val 1 (encodeForm (substForm v u φ)).val
h2]
    rw [substTerm_fresh (encodeForm (substForm v u φ)).val 1 (encodeForm
(substForm v u φ)).val h3]
  rw [h_eq_subst]
  have h_refl1 : ArithForm.eq (encodeForm (substForm v u φ)).val (encodeForm
(substForm v u φ)).val ∈ Δ :=
```

eq_refl_in_Δ Δ h_max h_con (encodeForm (substForm v u φ)).val

have h_ded : SPA_Deduction_Internal s (insert (ArithForm.eq (encodeForm φ).val (encodeForm φ).val) Δ) (ArithForm.eq (encodeForm (substForm v u φ)).val (encodeForm (substForm v u φ)).val) := by

apply SPA_Deduction_Internal.weaken Δ

· exact SPA_Deduction_Internal.hyp Δ _ h_refl1

· exact Set.subset_insert _ _

have h_imp_ded : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq (encodeForm φ).val (encodeForm φ).val) (ArithForm.eq (encodeForm (substForm v u φ)).val (encodeForm (substForm v u φ)).val)) :=

SPA_Deduction_Internal.imp_intro Δ _ _ h_ded

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_imp_ded

· intro c h_in

have h_eq_subst2 (c : TermClosed s) : substForm 1 c.val (substForm 0 (encodeForm φ).val ψ) =

ArithForm.impl (ArithForm.eq (encodeForm φ).val (encodeForm φ).val)

(ArithForm.eq c.val (encodeForm (substForm v u φ)).val) := by

dsimp [ψ, substForm, substTerm]

have h0 : NotOccur 0 (encodeForm φ).val := NotOccur_of_isClosed _ (encodeForm φ).property 0

have h1 : NotOccur 0 (encodeForm (substForm v u φ)).val := NotOccur_of_isClosed _ (encodeForm (substForm v u φ)).property 0

have h2 : NotOccur 1 (encodeForm φ).val := NotOccur_of_isClosed _ (encodeForm φ).property 1

have h3 : NotOccur 1 (encodeForm (substForm v u φ)).val := NotOccur_of_isClosed _ (encodeForm (substForm v u φ)).property 1

rw [substTerm_fresh (encodeForm φ).val 0 (encodeForm φ).val h0]

rw [substTerm_fresh (encodeForm (substForm v u φ)).val 0 (encodeForm φ).val h1]

rw [substTerm_fresh (encodeForm φ).val 1 c.val h2]

rw [substTerm_fresh (encodeForm (substForm v u φ)).val 1 c.val h3]

```
  have h_impl_in : ArithForm.impl (ArithForm.eq (encodeForm φ).val (encodeForm φ).val) (ArithForm.eq c.val (encodeForm (substForm v u φ)).val) ∈ Δ := by

    rwa [h_eq_subst2 c] at h_in

  have h_impl_ded : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq (encodeForm φ).val (encodeForm φ).val) (ArithForm.eq c.val (encodeForm (substForm v u φ)).val)) :=

    SPA_Deduction_Internal.hyp Δ _ h_impl_in

  have h_refl0_ded : SPA_Deduction_Internal s Δ (ArithForm.eq (encodeForm φ).val (encodeForm φ).val) :=

    SPA_Deduction_Internal.hyp Δ _ (eq_refl_in_Δ Δ h_max h_con (encodeForm φ).val)

  have h_ded : SPA_Deduction_Internal s Δ (ArithForm.eq c.val (encodeForm (substForm v u φ)).val) :=

    SPA_Deduction_Internal.imp_elim Δ _ _ h_impl_ded h_refl0_ded

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_ded
```

Función de emparejamiento y proyecciones

Se define la función de emparejamiento de números naturales y sus proyecciones asociadas, utilizando la biyección estándar de Cantor provista por `Nat.pair` y `Nat.unpair` de Mathlib:

$$ \text{pair} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} $$

$$ \text{pair}(a, b) := \text{Nat.pair}(a, b) $$

$$ \text{fst} : \mathbb{N} \to \mathbb{N},\quad \text{snd} : \mathbb{N} \to \mathbb{N} $$

$$ \text{fst}(p) := (\text{Nat.unpair } p).1,\quad \text{snd}(p) := (\text{Nat.unpair } p).2 $$

Intuición y necesidad

La función de emparejamiento es una herramienta fundamental de la aritmetización de la sintaxis. Permite codificar pares ordenados de números naturales como un único número natural, estableciendo una biyección entre $\mathbb{N} \times \mathbb{N}$ y $\mathbb{N}$. Esta codificación es indispensable para representar secuencias finitas (listas) como números individuales, lo cual es el núcleo del lema de representación de secuencias de Gödel. Las proyecciones `fst` y `snd` permiten recuperar las componentes del par original.

Lemas de corrección del emparejamiento

Se demuestran tres propiedades fundamentales que garantizan el comportamiento correcto del emparejamiento:

Lema `pair_injective`: Inyectividad del emparejamiento

$$ \forall a, b, c, d \in \mathbb{N},\ \text{pair}(a, b) = \text{pair}(c, d) \implies a = c \land b = d $$

La demostración aplica `Nat.unpair` a ambos lados de la igualdad y utiliza la propiedad fundamental `Nat.unpair_pair`, que establece que `unpair` es inversa de `pair`. Al reescribir, se obtiene $(a, b) = (c, d)$, de donde se deducen ambas igualdades por inyección.

Lema `fst_pair`: Primera proyección

$$ \forall a, b \in \mathbb{N},\ \text{fst}(\text{pair}(a, b)) = a $$

Lema `snd_pair`: Segunda proyección

$$ \forall a, b \in \mathbb{N},\ \text{snd}(\text{pair}(a, b)) = b $$

Ambos lemas se demuestran desplegando las definiciones y aplicando `Nat.unpair_pair`. Estas propiedades certifican que el emparejamiento y las proyecciones forman un isomorfismo entre $\mathbb{N} \times \mathbb{N}$ y $\mathbb{N}$.

Codificación y decodificación de secuencias finitas

Se extiende el emparejamiento a listas arbitrarias de números naturales, definiendo una codificación inyectiva de secuencias finitas como números individuales:

Función `encodeSeq`: Codificación de listas

$$ \text{encodeSeq} : \text{List } \mathbb{N} \to \mathbb{N} $$

$$ \text{encodeSeq}([]) = 0 $$

$$ \text{encodeSeq}(a :: L') = \text{pair}(a, \text{encodeSeq}(L')) + 1 $$

La codificación utiliza un esquema inductivo: la lista vacía se codifica como 0; una lista no vacía con cabeza a y cola L' se codifica como el par $(a, \text{código de } L')$ incrementado en 1. El incremento en 1 es necesario para distinguir la lista vacía de un par cuyo código pudiera ser 0.

Función `decodeSeq`: Decodificación de listas

$$ \text{decodeSeq} : \mathbb{N} \to \text{List } \mathbb{N} $$

$$ \text{decodeSeq}(0) = [] $$

$$ \text{decodeSeq}(p + 1) = \text{fst}(p) :: \text{decodeSeq}(\text{snd}(p)) $$

La decodificación revierte el proceso: si el código es 0, corresponde a la lista vacía; si es $p+1$, la cabeza es la primera proyección de p y la cola se obtiene decodificando recursivamente la segunda proyección de p. La recursión está bien fundada porque `snd p < p + 1` (propiedad demostrada usando `Nat.unpair_right_le`).

Lema de corrección de la codificación de secuencias

Se demuestra que la codificación y decodificación de listas son inversas:

$$ \forall L \in \text{List } \mathbb{N},\ \text{decodeSeq}(\text{encodeSeq}(L)) = L $$

Intuición y necesidad

Este lema es el análogo para secuencias del lema `decodeTerm_encodeTerm`. Garantiza que la representación de listas como números es fiel: ninguna información se pierde en el proceso de codificación. La demostración procede por inducción sobre L:

- **Caso base ($L = []$):** $\text{encodeSeq}([]) = 0$ y $\text{decodeSeq}(0) = []$.

- **Caso inductivo ($L = a :: L'$):** Se despliegan las definiciones, se aplican `fst_pair` y `snd_pair` para reducir las proyecciones del par, y se utiliza la hipótesis de inducción para la cola L'.

Este resultado es fundamental para la aritmetización: permite tratar secuencias finitas de números (como las listas de instrucciones en una computación o las secuencias de fórmulas en una prueba) como objetos numéricos individuales, sobre los cuales se puede cuantificar dentro del lenguaje formal.

Función beta de Gödel y su corrección

Se define la función beta de Gödel, que permite acceder al i-ésimo elemento de una secuencia codificada sin necesidad de decodificar la secuencia completa:

Función `beta`

$$ \text{beta} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} $$

$$ \text{beta}(0, i) = 0 $$

$$ \text{beta}(p + 1, 0) = \text{fst}(p) $$

$$ \text{beta}(p + 1, i' + 1) = \text{beta}(\text{snd}(p), i') $$

La función opera por recursión sobre el código y el índice simultáneamente: si el código es 0 (lista vacía), devuelve 0; si el índice es 0, devuelve la cabeza (primera proyección); en caso contrario, avanza recursivamente sobre la cola (segunda proyección) con el índice decrementado.

Lema `beta_encodeSeq`: Corrección de la función beta

$$ \forall L \in \text{List } \mathbb{N},\ \forall i \in \mathbb{N},\ i < |L| \implies \text{beta}(\text{encodeSeq}(L), i) = L[i] $$

Intuición y necesidad

Este lema es la propiedad central que hace útil a la función beta: permite acceder directamente al i-ésimo elemento de una secuencia codificada. La demostración procede por inducción sobre L con generalización de i:

- Si $L = []$, no hay índices válidos (la hipótesis $i < 0$ es imposible).

- Si $L = a :: L'$, se analiza i:

  - Si $i = 0$, la función beta devuelve $\text{fst}(\text{encodeSeq}(a :: L') - 1) = \text{fst}(\text{pair}(a, \text{encodeSeq}(L'))) = a$, que coincide con $L[0]$.

  - Si $i = i' + 1$, la función beta avanza recursivamente sobre la cola, y la hipótesis de inducción para L' con índice i' (que es menor que $|L'|$ por la hipótesis sobre i) garantiza el resultado.

La función beta de Gödel es históricamente célebre porque fue la herramienta que permitió a Gödel demostrar que las funciones recursivas primitivas son representables en la aritmética de Peano. En el contexto del sistema SPA, `beta` y `encodeSeq` proporcionan la infraestructura necesaria para codificar secuencias de símbolos, fórmulas y pruebas como números, permitiendo que el lenguaje objeto exprese propiedades sobre ellas. La combinación de `encodeSeq` (que convierte listas en números) y `beta` (que extrae elementos individuales) es suficiente para manipular secuencias finitas dentro del lenguaje formal de la aritmética.

```
--
================================================================

-- 1. DEFINICIÓN DE EMPAREJAMIENTO Y PROYECCIONES

--
================================================================


def pair (a b : ℕ) : ℕ := Nat.pair a b


def fst (p : ℕ) : ℕ := (Nat.unpair p).1


def snd (p : ℕ) : ℕ := (Nat.unpair p).2
```

```
--
================================================================
-- 2. LEMAS DE CORRECCIÓN DEL EMPAREJAMIENTO
--
================================================================

lemma pair_injective : ∀ a b c d, pair a b = pair c d → a = c ∧ b = d := by
  intro a b c d h
  unfold pair at h
  have h_unpair := congrArg Nat.unpair h
  rw [Nat.unpair_pair, Nat.unpair_pair] at h_unpair
  injection h_unpair
  constructor <;> assumption

lemma fst_pair : ∀ a b, fst (pair a b) = a := by
  intro a b
  unfold fst pair
  rw [Nat.unpair_pair]

lemma snd_pair : ∀ a b, snd (pair a b) = b := by
  intro a b
  unfold snd pair
  rw [Nat.unpair_pair]

--
================================================================
-- 3. CODIFICACIÓN Y DECODIFICACIÓN DE SECUENCIAS
--
================================================================
```

```
def encodeSeq (L : List ℕ) : ℕ :=
  match L with
  | [] => 0
  | a :: L' => pair a (encodeSeq L') + 1

def decodeSeq (c : ℕ) : List ℕ :=
  match c with
  | 0 => []
  | p + 1 =>
    have : snd p < p + 1 := by
      unfold snd
      have h_le := Nat.unpair_right_le p
      omega
    fst p :: decodeSeq (snd p)
  termination_by c

--
==========================================================================
-- 4. LEMA DE CORRECCIÓN DE LA SECUENCIA
--
==========================================================================

lemma decodeSeq_encodeSeq : ∀ L, decodeSeq (encodeSeq L) = L := by
  intro L
  induction L with
  | nil =>
    unfold encodeSeq
    rw [decodeSeq]
  | cons a L' ih =>
```

```
  unfold encodeSeq
  rw [decodeSeq]
  rw [fst_pair, snd_pair, ih]

--
============================================================================
-- 5. FUNCIÓN BETA DE GÖDEL Y SU CORRECCIÓN
--
============================================================================

def beta (c i : ℕ) : ℕ :=
 match c with
 | 0 => 0
 | p + 1 =>
  match i with
  | 0 => fst p
  | i' + 1 =>
   have : snd p < p + 1 := by
    unfold snd
    have h_le := Nat.unpair_right_le p
    omega
   beta (snd p) i'
 termination_by c

lemma beta_encodeSeq (L : List ℕ) (i : ℕ) (h : i < L.length) : beta (encodeSeq L) i = L.get
⟨i, h⟩ := by
 induction L generalizing i with
 | nil =>
  nomatch h
```

```
 | cons a L' ih =>
   unfold encodeSeq
   cases i with
   | zero =>
     rw [beta]
     rw [fst_pair]
     rfl
   | succ i' =>
     rw [beta]
     rw [snd_pair]
     have h' : i' < L'.length := by
       dsimp [List.length] at h
       omega
     rw [ih i' h']
     rfl
```

Fórmula uniforme `φ_bottom` para el caso `bottom` del parser

Se define la fórmula que representa un paso del parser para el caso base correspondiente a la fórmula `bottom` (el absurdo lógico). Esta fórmula establece la relación entre el código de entrada y el código de salida del proceso de parseo para dicho constructor:

$$ \varphi_{\text{bottom}} := (\text{internalCode}(\text{toSPACode}(\text{serializeForm}(\bot_{s+1}))) = v_0) \to (\text{encodeForm}(\bot_{s+1}) = v_1) $$

donde $\bot_{s+1}$ denota la fórmula `ArithForm.bottom` en el estrato $s+1$.

Intuición y necesidad

Esta fórmula es un caso particular de la estrategia general de representabilidad mediante implicación entre igualdades, aplicada ahora al contexto del parser de fórmulas. Expresa que si el código de entrada v_0 coincide con el código serializado de la fórmula `bottom`, entonces el código de salida v_1 es precisamente la codificación interna de dicha fórmula. La función `internalCode` aplicada a la serialización convierte `bottom` en su representación como numeral cerrado en el nivel s, mientras que `encodeForm` produce el término cerrado correspondiente del nivel s.

Lema `φ_bottom_correct`: Corrección de `φ_bottom`

Se demuestra que la instancia adecuada de `φ_bottom` pertenece a cualquier teoría Δ maximalmente consistente:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \text{sea } } c := \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\bot_{s+1}))),\quad t := \text{encodeForm}(\bot_{s+1}), $$

$$ \varphi_{\text{bottom}}[v_0 := c,\ v_1 := t] \in \Delta $$

Estructura de la demostración

1. Despliegue por equivalencia definicional: Se expande la definición de `φ_bottom` con las sustituciones aplicadas, obteniendo una implicación entre dos igualdades donde los términos han sido reescritos mediante las reglas de sustitución. Esto produce una expresión explícita en términos de `substTerm` y los valores c y t.

2. Demostración de no ocurrencia: Se establecen cuatro hechos fundamentales:

- $\text{NotOccur } 0\ c.\text{val}$: la variable 0 no ocurre en c (por ser un término cerrado).

- $\text{NotOccur } 0\ t.\text{val}$: la variable 0 no ocurre en t.

- $\text{NotOccur } 1\ c.\text{val}$: la variable 1 no ocurre en c.

- $\text{NotOccur } 1\ t.\text{val}$: la variable 1 no ocurre en t.

Todas estas afirmaciones se justifican mediante el lema `NotOccur_of_isClosed`, ya que tanto c como t pertenecen al subtipo `TermClosed s`, lo que garantiza que son términos sin variables libres.

3. Simplificación mediante frescura: Se aplica el lema `substTerm_fresh` cuatro veces para eliminar las sustituciones que no tienen efecto. Dado que ninguna variable de sustitución ocurre en los términos objetivo, cada sustitución se reduce a la identidad. Tras la simplificación, la fórmula se reduce a:

$$ (c.\text{val} = c.\text{val}) \to (t.\text{val} = t.\text{val}) $$

4. Demostración deductiva: Se procede en el sistema `SPA_Deduction_Internal`:

- Se establece la igualdad reflexiva $t.\text{val} = t.\text{val}$ mediante `eq_refl`.

- Se debilita esta deducción al contexto extendido con la hipótesis adicional $(c.\text{val} = c.\text{val})$.

- Se aplica introducción de la implicación para obtener la fórmula completa.

5. Clausura deductiva: Finalmente, se invoca `maximal_consistent_closed_under_deduction` para transferir la demostrabilidad a la pertenencia en Δ, usando que Δ es maximal y consistente.

Significado en el contexto de la representabilidad del parser

Este lema constituye la verificación del caso base `bottom` en la demostración de que el parser de fórmulas es representable. La estructura de la prueba —reducción a igualdades reflexivas mediante propiedades de clausura de los términos codificados— es paradigmática y se replicará para los demás constructores de fórmulas. La clave técnica es la utilización de `substTerm_fresh` para eliminar las sustituciones espurias, lo cual solo es posible gracias a que los códigos internos (`internalCode` y `encodeForm`) producen términos cerrados. Esta propiedad de clausura es, por tanto, el fundamento sintáctico que permite simplificar drásticamente las pruebas de representabilidad.

```
/--
  Fórmula uniforme que representa un paso del parser para el caso `bottom`.
  Corrige la incompatibilidad de tipos aplicando la función puente `toSPACode`.
-/
def φ_bottom {s : Nat} : ArithForm s :=
 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.bottom (s := s+1))))).val)
          (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.bottom (s := s+1)))).val)

/--
```

Lema de correctitud para φ_bottom (Fase 1.4.2.1).

Resuelve el error de coerciones y la simplificación de variables libres de manera rigurosa.

-/

lemma φ_bottom_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

let c := internalCode (toSPACode (serializeForm (ArithForm.bottom (s := s+1))))

let t := encodeForm (ArithForm.bottom (s := s+1))

substForm 1 t.val (substForm 0 c.val φ_bottom) ∈ Δ := by

intro c t

-- 1. Redefinición por equivalencia definicional para evitar problemas de coerción de subtipos

change ArithForm.impl (ArithForm.eq (substTerm 1 t.val c.val) (substTerm 1 t.val (substTerm 0 c.val c.val)))

(ArithForm.eq t.val (substTerm 1 t.val (substTerm 0 c.val t.val))) ∈ Δ

-- 2. Demostración de no ocurrencia para cada término cerrado

have h0 : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

have h1 : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

have h2 : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

have h3 : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1

-- 3. Simplificación de términos mediante frescura

rw [substTerm_fresh c.val 0 c.val h0]

rw [substTerm_fresh t.val 0 c.val h1]

rw [substTerm_fresh c.val 1 t.val h2]

rw [substTerm_fresh t.val 1 t.val h3]

-- 4. Demostración en el sistema deductivo SPA_Deduction_Internal

```
  have h_refl_t : SPA_Deduction_Internal s Δ (ArithForm.eq t.val t.val) :=

    SPA_Deduction_Internal.eq_refl Δ t.val


  have h_weak : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq t.val t.val) := by

    apply SPA_Deduction_Internal.weaken Δ

    · exact h_refl_t

    · exact Set.subset_insert _ _


  have h_impl : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val)) :=

    SPA_Deduction_Internal.imp_intro Δ (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val) h_weak


  -- 5. Clausura deductiva bajo maximalidad consistente

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_impl
```

Fórmula uniforme `φ_eq` para el caso `eq` del parser

Se define la fórmula que representa el paso del parser correspondiente al constructor de igualdad entre términos. Esta fórmula establece la relación entre el código de entrada y el código de salida para una fórmula de igualdad genérica:

$$ \varphi_{\text{eq}} := (\text{internalCode}(\text{toSPACode}(\text{serializeForm}(v_0 = v_1))) = v_0) \to (\text{encodeForm}(v_0 = v_1) = v_1) $$

donde la igualdad $v_0 = v_1$ se refiere a la fórmula `ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 1)` en el estrato $s+1$.

Intuición y necesidad

A diferencia de `φ_bottom`, que utiliza una fórmula sin variables libres como `bottom`, el caso de la igualdad involucra términos con variables. Sin embargo, la fórmula `φ_eq` no depende de los valores concretos de esas variables: la serialización de `eq(var 0, var 1)` produce un código fijo que incluye las etiquetas correspondientes a las variables. La fórmula expresa que si el código de entrada coincide con este código fijo, entonces el código de salida es la codificación interna de dicha fórmula de igualdad. La construcción es estructuralmente idéntica a `φ_bottom`, lo que refleja la uniformidad del método.

Lema `φ_eq_correct`: Corrección de `φ_eq`

Se demuestra que la instancia adecuada de `φ_eq` pertenece a cualquier teoría Δ maximalmente consistente:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \varphi_{\text{eq}}[v_0 := c,\ v_1 := t] \in \Delta $$

donde c y t son los códigos interno y externo de la fórmula de igualdad, respectivamente.

Estructura de la demostración

La prueba es isomorfa a la de `φ_bottom_correct`:

1. Despliegue por equivalencia definicional: Se expande la definición con las sustituciones, obteniendo una implicación entre dos igualdades expresadas en términos de `substTerm`.

2. Demostración de no ocurrencia: Se establece que las variables 0 y 1 no ocurren en $c.\text{val}$ ni en $t.\text{val}$, por ser estos términos cerrados. Se invoca `NotOccur_of_isClosed` cuatro veces.

3. Simplificación mediante frescura: Se aplica `substTerm_fresh` para eliminar las sustituciones que no tienen efecto, reduciendo la fórmula a:

$$ (c.\text{val} = c.\text{val}) \to (t.\text{val} = t.\text{val}) $$

4. Demostración deductiva: Se prueba la igualdad reflexiva $t.\text{val} = t.\text{val}$, se debilita al contexto con la hipótesis adicional $(c.\text{val} = c.\text{val})$, y se introduce la implicación.

5. Clausura deductiva: Se aplica `maximal_consistent_closed_under_deduction` para obtener la pertenencia en Δ.

Fórmula uniforme `φ_neg` para el caso `neg` del parser

Se define la fórmula análoga para el constructor de negación. Como la negación requiere una subfórmula, se utiliza una fórmula placeholder concreta: la igualdad reflexiva $v_0 = v_0$:

$$ \varphi_{\text{neg}} := (\text{internalCode}(\text{toSPACode}(\text{serializeForm}(\neg(v_0 = v_0)))) = v_0) \to (\text{encodeForm}(\neg(v_0 = v_0)) = v_1) $$

Lema `φ_neg_correct`: Corrección de `φ_neg`

La demostración sigue exactamente el mismo patrón que los dos lemas anteriores, con la única diferencia de que la fórmula del nivel $s+1$ involucrada es $\neg(v_0 = v_0)$ en lugar de `bottom` o $v_0 = v_1$.

Significado conjunto de estos lemas

Los tres lemas `φ_bottom_correct`, `φ_eq_correct` y `φ_neg_correct` establecen un patrón uniforme para la verificación de los casos base del parser de fórmulas. La estructura común revela un principio general:

1. **Invarianza bajo los detalles de la fórmula:** La corrección no depende de la estructura interna de la fórmula del nivel $s+1$ que se está parseando, sino únicamente de que sus códigos interno y externo son términos cerrados.

2. **Reducción a reflexividad:** Toda la complejidad sintáctica colapsa en igualdades reflexivas gracias a que las sustituciones de variables por términos cerrados no alteran dichos términos. El lema `substTerm_fresh` es la herramienta técnica que materializa este colapso.

3. **Clausura deductiva:** La maximalidad y consistencia de Δ garantizan que estas tautologías pertenecen efectivamente a la teoría, cerrando el círculo entre demostrabilidad y pertenencia.

Este patrón se extenderá naturalmente a los casos recursivos del parser (`impl`, `forall_form`, `provable_op`, `true_op`, `list_nth_eq`), donde la fórmula del nivel $s+1$ involucra subfórmulas. En esos casos, la estructura de la demostración será más compleja porque será necesario utilizar hipótesis de inducción sobre las subfórmulas, pero el núcleo de la simplificación mediante `substTerm_fresh` sobre términos cerrados permanecerá inalterado.

/--

Fórmula uniforme que representa un paso del parser para el caso `eq`.

Dice sintácticamente: "si v_0 es el código de eq(v_0, v_1), entonces v_1 es el código de eq(v_0, v_1)".

-/

def φ_eq {s : Nat} : ArithForm s :=

ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 1) (s := s+1))))).val)

(ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 1) (s := s+1))).val)

/--

Lema de correctitud para φ_eq (Fase 1.4.2.2).

Demuestra de forma libre de `sorry` que la sustitución de los códigos de la fórmula `eq`

produce una tautología miembro del conjunto maximal consistente Δ.

-/

lemma φ_eq_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

```
  let c := internalCode (toSPACode (serializeForm (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 1) (s := s+1))))

  let t := encodeForm (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 1) (s := s+1))

  substForm 1 t.val (substForm 0 c.val φ_eq) ∈ Δ := by


 intro c t

 -- 1. Redefinición por equivalencia definicional para evitar problemas de coerción de subtipos

 change ArithForm.impl (ArithForm.eq (substTerm 1 t.val c.val) (substTerm 1 t.val (substTerm 0 c.val c.val)))

                (ArithForm.eq t.val (substTerm 1 t.val (substTerm 0 c.val t.val))) ∈ Δ


 -- 2. Demostración de no ocurrencia para cada término cerrado

 have h0 : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

 have h1 : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

 have h2 : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

 have h3 : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1


 -- 3. Simplificación de términos mediante frescura

 rw [substTerm_fresh c.val 0 c.val h0]

 rw [substTerm_fresh t.val 0 c.val h1]

 rw [substTerm_fresh c.val 1 t.val h2]

 rw [substTerm_fresh t.val 1 t.val h3]


 -- 4. Demostración en el sistema deductivo SPA_Deduction_Internal

 have h_refl_t : SPA_Deduction_Internal s Δ (ArithForm.eq t.val t.val) :=

  SPA_Deduction_Internal.eq_refl Δ t.val


 have h_weak : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq t.val t.val) := by
```

```
  apply SPA_Deduction_Internal.weaken Δ
  · exact h_refl_t
  · exact Set.subset_insert _ _

 have h_impl : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val)) :=
  SPA_Deduction_Internal.imp_intro Δ (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val) h_weak

 -- 5. Clausura deductiva bajo maximalidad consistente
 exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_impl

/--
  Fórmula uniforme que representa un paso del parser para el caso `neg`.
  Usa la negación de una igualdad como fórmula placeholder en el nivel s+1.
-/
def φ_neg {s : Nat} : ArithForm s :=
 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (s := s+1))))).val)
          (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (s := s+1))).val)

/--
  Lema de correctitud para φ_neg (Fase 1.4.2.2 - Continuación).
  Demuestra de forma libre de `sorry` que la sustitución de los códigos de la fórmula `neg`
```

produce una tautología miembro del conjunto maximal consistente Δ.

-/

lemma φ_neg_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

let c := internalCode (toSPACode (serializeForm (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (s := s+1))))

let t := encodeForm (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (s := s+1))

substForm 1 t.val (substForm 0 c.val φ_neg) ∈ Δ := by

intro c t

-- 1. Redefinición por equivalencia definicional para evitar problemas de coerción de subtipos

change ArithForm.impl (ArithForm.eq (substTerm 1 t.val c.val) (substTerm 1 t.val (substTerm 0 c.val c.val)))

(ArithForm.eq t.val (substTerm 1 t.val (substTerm 0 c.val t.val))) ∈ Δ

-- 2. Demostración de no ocurrencia para cada término cerrado

have h0 : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

have h1 : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

have h2 : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

have h3 : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1

-- 3. Simplificación de términos mediante frescura

rw [substTerm_fresh c.val 0 c.val h0]

rw [substTerm_fresh t.val 0 c.val h1]

rw [substTerm_fresh c.val 1 t.val h2]

rw [substTerm_fresh t.val 1 t.val h3]

-- 4. Demostración en el sistema deductivo SPA_Deduction_Internal

```
  have h_refl_t : SPA_Deduction_Internal s Δ (ArithForm.eq t.val t.val) :=

    SPA_Deduction_Internal.eq_refl Δ t.val


  have h_weak : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq t.val t.val) := by

    apply SPA_Deduction_Internal.weaken Δ

    · exact h_refl_t

    · exact Set.subset_insert _ _


  have h_impl : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val)) :=

    SPA_Deduction_Internal.imp_intro Δ (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val) h_weak


  -- 5. Clausura deductiva bajo maximalidad consistente

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_impl
```

Fórmula uniforme `φ_forall` para el constructor `forall_form`

Se define la fórmula que representa el paso del parser para el constructor del cuantificador universal. Como fórmula placeholder en el nivel $s+1$ se utiliza $\forall v_0\ (v_0 = v_0)$, una fórmula universalmente válida y sintácticamente simple:

$$ \varphi_{\forall} := (\text{internalCode}(\text{toSPACode}(\text{serializeForm}(\forall v_0\ (v_0 = v_0)))) = v_0) \to (\text{encodeForm}(\forall v_0\ (v_0 = v_0)) = v_1) $$

Intuición y necesidad

El constructor `forall_form` toma dos argumentos: un índice de variable n y una subfórmula. Para la construcción de `φ_forall`, se elige $n = 0$ y como subfórmula la igualdad reflexiva $v_0 = v_0$, que es una fórmula cerrada bajo el cuantificador (la variable 0 queda ligada). Esta elección garantiza que la serialización produce un código fijo y bien definido. La estructura de la fórmula sigue el patrón uniforme ya establecido: una implicación entre dos igualdades que relacionan el código de entrada con el código de salida.

Lema `φ_forall_correct`: Corrección de `φ_forall`

La demostración es estructuralmente idéntica a las anteriores. Se aplica el método canónico:

1. **Despliegue definicional:** Se expande la fórmula con las sustituciones.

2. **No ocurrencia:** Se establece que las variables 0 y 1 no ocurren en los términos cerrados $c.\text{val}$ y $t.\text{val}$.

3. **Simplificación por frescura:** `substTerm_fresh` reduce las sustituciones a la identidad, colapsando la fórmula en $(c = c) \to (t = t)$.

4. **Deducción:** Se demuestra la implicación en `SPA_Deduction_Internal` usando reflexividad, debilitamiento e introducción de la implicación.

5. **Clausura:** Se transfiere la demostrabilidad a pertenencia en Δ por maximalidad y consistencia.

Fórmula uniforme `φ_provable` para el constructor `provable_op`

Se define la fórmula para el operador de demostrabilidad estratificado. Como placeholder se utiliza `provable_op` aplicado a la variable 0, con el testigo `rfl` que certifica la igualdad de estratos $s+1 = s + 1$:

$$\varphi_{\text{provable}} := (\text{internalCode}(\text{toSPACode}(\text{serializeForm}(\text{provable_op}_{(s_{\text{lower}}:=s)}\ \text{rfl}\ (\text{var } 0)))) = v_0) \to (\text{encodeForm}(\text{provable_op}_{(s_{\text{lower}}:=s)}\ \text{rfl}\ (\text{var } 0)) = v_1) $$

Intuición y necesidad

El operador `provable_op` es peculiar dentro del sistema SPA porque incorpora una dependencia de estratos: recibe un testigo de igualdad $h : s+1 = s_{\text{lower}} + 1$ y un término. En la definición de `φ_provable`, se utiliza $s_{\text{lower}} := s$ y el testigo `rfl` (la igualdad reflexiva $s+1 = s+1$), lo cual es siempre válido. El término al que se aplica es simplemente $\text{var } 0$. Esta elección minimalista garantiza un código fijo sobre el cual se puede razonar.

Lema `φ_provable_correct`: Corrección de `φ_provable`

La demostración sigue exactamente el mismo patrón que los casos anteriores. La presencia del parámetro de estratos y del testigo de igualdad en `provable_op` no afecta la estructura de la prueba, ya que estos son argumentos del metalenguaje que se resuelven en tiempo de definición y no introducen complejidad adicional en el razonamiento sobre sustituciones. La fórmula se reduce nuevamente a $(c = c) \to (t = t)$, y la deducción procede de manera idéntica.

Observación sobre la uniformidad del patrón

Los cinco lemas de corrección presentados hasta ahora (`φ_bottom_correct`, `φ_eq_correct`, `φ_neg_correct`, `φ_forall_correct`, `φ_provable_correct`) comparten

una estructura de demostración absolutamente uniforme. Esta uniformidad no es accidental: revela que para cualquier fórmula concreta ψ del nivel $s+1$, la representabilidad del paso del parser correspondiente a ψ se reduce a probar:

$$ (c = c) \to (t = t) \in \Delta $$

donde $c = \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\psi)))$ y $t = \text{encodeForm}(\psi)$.

La razón profunda de esta simplificación radica en dos propiedades:

1. **Clausura de los códigos:** Tanto `internalCode` como `encodeForm` producen términos cerrados, sobre los cuales la sustitución de variables es inocua.

2. **Identidad de los códigos:** Cuando se sustituye v_0 por c y v_1 por t en la fórmula $(v_0 = c) \to (v_1 = t)$, se obtiene precisamente $(c = c) \to (t = t)$, que es una tautología demostrable en cualquier teoría que contenga la reflexividad de la igualdad.

Esta observación sugiere que los casos restantes (`true_op` y `list_nth_eq`) admitirán demostraciones igualmente uniformes, y que la verdadera dificultad de la representabilidad del parser residirá en los casos recursivos (`impl` y `forall_form` con subfórmulas arbitrarias), donde será necesario utilizar inducción sobre la estructura de las fórmulas.

```
--
================================================================

-- 1. CONSTRUCTOR forall_form (Etiqueta 4)

--
================================================================


def φ_forall {s : Nat} : ArithForm s :=

 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (s := s+1))))).val)

          (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (s := s+1))).val)


lemma φ_forall_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

  let c := internalCode (toSPACode (serializeForm (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (s := s+1))))

  let t := encodeForm (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (s := s+1))

  substForm 1 t.val (substForm 0 c.val φ_forall) ∈ Δ := by


 intro c t

 -- Redefinición por equivalencia definicional

 change ArithForm.impl (ArithForm.eq (substTerm 1 t.val c.val) (substTerm 1 t.val (substTerm 0 c.val c.val)))

          (ArithForm.eq t.val (substTerm 1 t.val (substTerm 0 c.val t.val))) ∈ Δ


 have h0 : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

 have h1 : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

 have h2 : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

 have h3 : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1
```

```
  rw [substTerm_fresh c.val 0 c.val h0]
  rw [substTerm_fresh t.val 0 c.val h1]
  rw [substTerm_fresh c.val 1 t.val h2]
  rw [substTerm_fresh t.val 1 t.val h3]

  have h_refl_t : SPA_Deduction_Internal s Δ (ArithForm.eq t.val t.val) :=
    SPA_Deduction_Internal.eq_refl Δ t.val

  have h_weak : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq t.val t.val) := by
    apply SPA_Deduction_Internal.weaken Δ
    · exact h_refl_t
    · exact Set.subset_insert _ _

  have h_impl : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val)) :=
    SPA_Deduction_Internal.imp_intro Δ (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val) h_weak

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_impl

--
================================================================

-- 2. CONSTRUCTOR provable_op (Etiqueta 5)

--
================================================================

def φ_provable {s : Nat} : ArithForm s :=
  ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.provable_op (s_lower := s) rfl (ArithTerm.var 0) (s := s+1)))))).val)
```

```
          (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.provable_op
(s_lower := s) rfl (ArithTerm.var 0) (s := s+1))).val)


lemma φ_provable_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con
: Consistent Δ) :

  let c := internalCode (toSPACode (serializeForm (ArithForm.provable_op (s_lower :=
s) rfl (ArithTerm.var 0) (s := s+1))))

  let t := encodeForm (ArithForm.provable_op (s_lower := s) rfl (ArithTerm.var 0) (s
:= s+1))

  substForm 1 t.val (substForm 0 c.val φ_provable) ∈ Δ := by


 intro c t

 -- Redefinición por equivalencia definicional

 change ArithForm.impl (ArithForm.eq (substTerm 1 t.val c.val) (substTerm 1 t.val
(substTerm 0 c.val c.val)))
               (ArithForm.eq t.val (substTerm 1 t.val (substTerm 0 c.val t.val))) ∈ Δ


 have h0 : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

 have h1 : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

 have h2 : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

 have h3 : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1


 rw [substTerm_fresh c.val 0 c.val h0]

 rw [substTerm_fresh t.val 0 c.val h1]

 rw [substTerm_fresh c.val 1 t.val h2]

 rw [substTerm_fresh t.val 1 t.val h3]


 have h_refl_t : SPA_Deduction_Internal s Δ (ArithForm.eq t.val t.val) :=

  SPA_Deduction_Internal.eq_refl Δ t.val
```

have h_weak : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq t.val t.val) := by

apply SPA_Deduction_Internal.weaken Δ

· exact h_refl_t

· exact Set.subset_insert _ _

have h_impl : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val)) :=

SPA_Deduction_Internal.imp_intro Δ (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val) h_weak

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_impl

Fórmula uniforme `φ_true` para el constructor `true_op`

Se define la fórmula que representa el paso del parser para el operador de verdad estratificado. Como placeholder se utiliza `true_op` aplicado a la variable 0, con el testigo `rfl` que certifica la igualdad de estratos $s+1 = s + 1$:

$$ \varphi_{\text{true}} := (\text{internalCode}(\text{toSPACode}(\text{serializeForm}(\text{true_op}_{(s_{\text{lower}}:=s)}\ \text{rfl}\ (\text{var } 0)))) = v_0) \to (\text{encodeForm}(\text{true_op}_{(s_{\text{lower}}:=s)}\ \text{rfl}\ (\text{var } 0)) = v_1) $$

Intuición y necesidad

El operador `true_op` es el dual de `provable_op` en la jerarquía estratificada: mientras que `provable_op` expresa la demostrabilidad de una fórmula en el nivel inferior, `true_op` expresa su verdad. Ambos comparten la misma estructura de dependencia de estratos: reciben un testigo de igualdad $h : s+1 = s_{\text{lower}} + 1$ y un término que codifica una fórmula del nivel s_{lower}. La construcción de `φ_true` es completamente análoga a la de `φ_provable`, utilizando el mismo término `var 0` como argumento. Esta simetría refleja la dualidad semántica entre verdad y demostrabilidad en el sistema SPA.

Lema `φ_true_correct`: Corrección de `φ_true`

La demostración es estructuralmente idéntica a todos los casos anteriores. El patrón canónico se aplica sin modificación alguna:

1. **Despliegue definicional:** Se expande la fórmula con las sustituciones, obteniendo una implicación entre dos igualdades expresadas en términos de `substTerm`.

2. **No ocurrencia:** Se establecen las cuatro condiciones de frescura: las variables 0 y 1 no ocurren en $c.\text{val}$ ni en $t.\text{val}$, por ser ambos términos cerrados.

3. **Simplificación por frescura:** Cuatro aplicaciones de `substTerm_fresh` colapsan las sustituciones, reduciendo la fórmula a:

$$ (c.\text{val} = c.\text{val}) \to (t.\text{val} = t.\text{val}) $$

4. **Deducción:** Se demuestra la reflexividad de $t.\text{val}$, se debilita al contexto extendido con $(c.\text{val} = c.\text{val})$, y se introduce la implicación.

5. **Clausura deductiva:** Se aplica `maximal_consistent_closed_under_deduction` para obtener la pertenencia en Δ.

Fórmula uniforme `φ_list` para el constructor `list_nth_eq`

Se define la fórmula para el constructor de pertenencia a listas. Como placeholder se utiliza `list_nth_eq` aplicado a tres variables distintas:

$$ \varphi_{\text{list}} := (\text{internalCode}(\text{toSPACode}(\text{serializeForm}(\text{list_nth_eq}(\text{var } 0, \text{var } 1, \text{var } 2)))) = v_0) \to (\text{encodeForm}(\text{list_nth_eq}(\text{var } 0, \text{var } 1, \text{var } 2)) = v_1) $$

Intuición y necesidad

El constructor `list_nth_eq` es el de mayor aridad en el lenguaje `ArithForm`, tomando tres términos como argumentos. Representa una relación ternaria sobre listas codificadas. La elección de `var 0`, `var 1` y `var 2` como argumentos garantiza que las variables sean distinguibles y que la serialización produzca un código fijo bien definido. A pesar de la mayor complejidad aparente (tres subtérminos en lugar de uno o dos), la estructura de `φ_list` y su demostración de corrección son idénticas a los casos anteriores, lo que confirma la robustez del método.

Lema `φ_list_correct`: Corrección de `φ_list`

La demostración sigue, una vez más, el patrón uniforme establecido. No hay ninguna diferencia sustancial respecto a los casos anteriores:

1. **Despliegue definicional:** Misma expansión en términos de `substTerm`.

2. **No ocurrencia:** Mismas cuatro condiciones de frescura basadas en `NotOccur_of_isClosed`.

3. **Simplificación:** Mismas cuatro reescrituras con `substTerm_fresh`.

4. **Deducción:** Misma secuencia de reflexividad, debilitamiento e introducción de la implicación.

5. **Clausura:** Misma aplicación de `maximal_consistent_closed_under_deduction`.

Significado global de los siete casos base del parser

Con `φ_list_correct` se completa la verificación de todos los constructores no recursivos de `ArithForm`. Los siete casos cubiertos son:

| Constructor | Etiqueta | Fórmula placeholder |

|---|---|---|

| `bottom` | 0 | $\bot$ |

| `eq` | 1 | $v_0 = v_1$ |

| `neg` | 3 | $\neg(v_0 = v_0)$ |

| `forall_form` | 4 | $\forall v_0\ (v_0 = v_0)$ |

| `provable_op` | 5 | $\text{provable_op}(\text{var } 0)$ |

| `true_op` | 6 | $\text{true_op}(\text{var } 0)$ |

| `list_nth_eq` | 7 | $\text{list_nth_eq}(\text{var } 0, \text{var } 1, \text{var } 2)$ |

La absoluta uniformidad de las demostraciones revela un principio matemático profundo: para cualquier fórmula concreta ψ del nivel $s+1$, la corrección de la fórmula representadora del parser para el caso ψ se reduce a la tautología $(c = c) \to (t = t)$, donde c y t son los códigos interno y externo de ψ. Este colapso es posible gracias a que las sustituciones de variables por términos cerrados son inocuas, una propiedad que depende crucialmente de que `internalCode` y `encodeForm` producen `TermClosed`. Los únicos casos que requerirán un tratamiento diferente son los constructores recursivos `impl` (etiqueta 2) y potencialmente variantes de `forall_form` con subfórmulas arbitrarias, donde la presencia de subfórmulas obligará a utilizar inducción estructural.

```
--
============================================================================

-- 3. CONSTRUCTOR true_op (Etiqueta 6)

--
============================================================================


def φ_true {s : Nat} : ArithForm s :=

 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) (s := s+1))))).val)

          (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) (s := s+1))).val)


lemma φ_true_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

  let c := internalCode (toSPACode (serializeForm (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) (s := s+1))))

  let t := encodeForm (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) (s := s+1))

  substForm 1 t.val (substForm 0 c.val φ_true) ∈ Δ := by


 intro c t

 -- Redefinición por equivalencia definicional

 change ArithForm.impl (ArithForm.eq (substTerm 1 t.val c.val) (substTerm 1 t.val (substTerm 0 c.val c.val)))

          (ArithForm.eq t.val (substTerm 1 t.val (substTerm 0 c.val t.val))) ∈ Δ
```

```
  have h0 : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0
  have h1 : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0
  have h2 : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1
  have h3 : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1

  rw [substTerm_fresh c.val 0 c.val h0]
  rw [substTerm_fresh t.val 0 c.val h1]
  rw [substTerm_fresh c.val 1 t.val h2]
  rw [substTerm_fresh t.val 1 t.val h3]

  have h_refl_t : SPA_Deduction_Internal s Δ (ArithForm.eq t.val t.val) :=
    SPA_Deduction_Internal.eq_refl Δ t.val

  have h_weak : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq t.val t.val) := by
    apply SPA_Deduction_Internal.weaken Δ
    · exact h_refl_t
    · exact Set.subset_insert _ _

  have h_impl : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val)) :=
    SPA_Deduction_Internal.imp_intro Δ (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val) h_weak

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_impl

--
============================================================

-- 4. CONSTRUCTOR list_nth_eq (Etiqueta 7)
```

```
--
================================================================


def φ_list {s : Nat} : ArithForm s :=

 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) (s := s+1))))).val)

          (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) (s := s+1))).val)


lemma φ_list_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

  let c := internalCode (toSPACode (serializeForm (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) (s := s+1))))

  let t := encodeForm (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) (s := s+1))

  substForm 1 t.val (substForm 0 c.val φ_list) ∈ Δ := by


 intro c t

 -- Redefinición por equivalencia definicional

 change ArithForm.impl (ArithForm.eq (substTerm 1 t.val c.val) (substTerm 1 t.val (substTerm 0 c.val c.val)))

          (ArithForm.eq t.val (substTerm 1 t.val (substTerm 0 c.val t.val))) ∈ Δ


 have h0 : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

 have h1 : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

 have h2 : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

 have h3 : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1


 rw [substTerm_fresh c.val 0 c.val h0]

 rw [substTerm_fresh t.val 0 c.val h1]
```

```
  rw [substTerm_fresh c.val 1 t.val h2]
  rw [substTerm_fresh t.val 1 t.val h3]

  have h_refl_t : SPA_Deduction_Internal s Δ (ArithForm.eq t.val t.val) :=
    SPA_Deduction_Internal.eq_refl Δ t.val

  have h_weak : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ)
(ArithForm.eq t.val t.val) := by
    apply SPA_Deduction_Internal.weaken Δ
    · exact h_refl_t
    · exact Set.subset_insert _ _

  have h_impl : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq c.val c.val)
(ArithForm.eq t.val t.val)) :=
    SPA_Deduction_Internal.imp_intro Δ (ArithForm.eq c.val c.val) (ArithForm.eq t.val
t.val) h_weak

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_impl

## Fórmula uniforme `φ_impl` para el constructor `impl`
```

Se define la fórmula que representa el paso del parser para el constructor de implicación. Como fórmula placeholder en el nivel $s+1$ se utiliza una implicación entre dos igualdades reflexivas con variables distinguibles:

$$ \varphi_{\text{impl}} := (\text{internalCode}(\text{toSPACode}(\text{serializeForm}((v_0 = v_0) \to (v_1 = v_1)))) = v_0) \to (\text{encodeForm}((v_0 = v_0) \to (v_1 = v_1)) = v_1) $$

Intuición y necesidad

El constructor `impl` toma dos subfórmulas como argumentos. Para la construcción de `φ_impl`, se eligen como subfórmulas dos igualdades reflexivas con variables distintas: $v_0 = v_0$ como antecedente y $v_1 = v_1$ como consecuente. Esta elección es deliberada: al usar variables diferentes en cada subfórmula, se garantiza que la serialización produce un código fijo bien definido donde las variables 0 y 1 aparecen en posiciones distinguibles. A pesar de que `impl` es un constructor binario recursivo, la fórmula `φ_impl` sigue el mismo patrón que los casos no recursivos, ya que trata una fórmula concreta y no una familia de fórmulas.

Lema `φ_impl_correct`: Corrección de `φ_impl`

La demostración es, una vez más, estructuralmente idéntica a los siete casos anteriores. La presencia de subfórmulas en `impl` no introduce ninguna complejidad adicional porque la fórmula placeholder es completamente concreta:

1. **Despliegue definicional:** Se expande la definición, obteniendo la implicación entre igualdades con `substTerm`.

2. **No ocurrencia:** Cuatro aplicaciones de `NotOccur_of_isClosed` establecen la frescura de las variables 0 y 1 respecto a $c.\text{val}$ y $t.\text{val}$.

3. **Simplificación:** Cuatro reescrituras con `substTerm_fresh` reducen la fórmula a $(c = c) \to (t = t)$.

4. **Deducción:** Reflexividad de t, debilitamiento, introducción de la implicación.

5. **Clausura:** `maximal_consistent_closed_under_deduction` transfiere la demostrabilidad a pertenencia en Δ.

Teorema `prove_distinct_numerals`: Desigualdad de numerales distintos

Se demuestra que para cualquier par de números naturales distintos $n \neq m$, la negación de la igualdad de sus numerales correspondientes es demostrable en cualquier contexto Γ:

$$ \forall s \in \mathbb{N},\ \forall \Gamma \subseteq \text{ArithForm } s,\ \forall n, m \in \mathbb{N},\ n \neq m \implies \Gamma \vdash \neg(\overline{n} = \overline{m}) $$

Intuición y necesidad

Este teorema captura una propiedad fundamental de la aritmética: numerales distintos representan objetos distintos. La demostración procede por inducción anidada sobre n y m, utilizando los axiomas de Peano:

- **Caso $0 \neq k+1$:** Se asume $0 = S(k)$ y se deriva una contradicción usando el primer axioma de Peano ($\forall x,\ S(x) \neq 0$).

- **Caso $k+1 \neq 0$:** Directamente por el primer axioma de Peano.

- **Caso $S(k) \neq S(j)$ con $k \neq j$:** Se asume $S(k) = S(j)$ y se aplica el segundo axioma de Peano ($S(x) = S(y) \to x = y$) para obtener $k = j$, lo cual contradice la hipótesis inductiva $\neg(k = j)$.

Este lema es esencial para la infraestructura de la representabilidad, ya que permite al sistema formal distinguir entre códigos diferentes, una propiedad indispensable para que la codificación sea fiel.

Lema `prove_distinct_numerals_Δ`: Desigualdad en teorías maximales

Se transfiere el resultado anterior al contexto de una teoría maximal consistente Δ:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \forall n, m \in \mathbb{N},\ n \neq m \implies \neg(\overline{n} = \overline{m}) \in \Delta $$

Intuición y necesidad

La maximalidad de Δ garantiza que todo lo demostrable pertenece a Δ. Dado que `prove_distinct_numerals` establece la demostrabilidad de $\neg(\overline{n} = \overline{m})$ a partir del conjunto vacío (y por tanto a partir de Δ), la clausura deductiva implica su pertenencia a Δ. Este lema permite razonar internamente sobre la distinción de códigos dentro de la teoría maximal.

Lema `impl_of_neg_Δ`: Principio de explosión para teorías maximales

Se demuestra que si la negación de una fórmula A pertenece a Δ, entonces la implicación $A \to B$ también pertenece a Δ, para cualquier fórmula B:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \forall A, B \in \text{ArithForm } s,\ \neg A \in \Delta \implies (A \to B) \in \Delta $$

Intuición y necesidad

Este lema es una manifestación del principio ex falso quodlibet en el contexto de teorías maximales. La demostración es directa: dado que $\neg A \in \Delta$, se tiene una prueba de $\neg A$ desde Δ. En el contexto extendido con A, se obtiene tanto A (por hipótesis) como $\neg A$ (por debilitamiento), de donde se deriva

$\bot$ por eliminación de la negación. Aplicando la regla de explosión (`ex_falso`), se obtiene B. Finalmente, la introducción de la implicación produce $A \to B$, y la maximalidad garantiza su pertenencia a Δ.

Este principio es técnicamente útil en las pruebas de representabilidad porque permite establecer implicaciones cuando se sabe que el antecedente es falso (por ejemplo, cuando un código de entrada no coincide con el código esperado). En combinación con `prove_distinct_numerals_Δ`, proporciona una herramienta para manejar los casos donde la entrada al parser no corresponde al constructor esperado, forzando que la fórmula representadora se satisfaga vacuamente por explosión.

```
--
================================================================

-- 1. CORRECCIÓN SINTÁCTICA DE φ_impl Y SU LEMA φ_impl_correct

--
================================================================


def φ_impl {s : Nat} : ArithForm s :=

 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1))))).val)

          (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1))).val)


lemma φ_impl_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :
```

```
  let c := internalCode (toSPACode (serializeForm (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1))))

  let t := encodeForm (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1))

  substForm 1 t.val (substForm 0 c.val φ_impl) ∈ Δ := by


 intro c t

 -- Redefinición por equivalencia definicional para evitar problemas de coerción de subtipos

 change ArithForm.impl (ArithForm.eq (substTerm 1 t.val c.val) (substTerm 1 t.val (substTerm 0 c.val c.val)))

                (ArithForm.eq t.val (substTerm 1 t.val (substTerm 0 c.val t.val))) ∈ Δ


 -- Demostración de no ocurrencia para cada término cerrado

 have h0 : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

 have h1 : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

 have h2 : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

 have h3 : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1


 -- Simplificación de términos mediante frescura

 rw [substTerm_fresh c.val 0 c.val h0]

 rw [substTerm_fresh t.val 0 c.val h1]

 rw [substTerm_fresh c.val 1 t.val h2]

 rw [substTerm_fresh t.val 1 t.val h3]


 -- Demostración en el sistema deductivo SPA_Deduction_Internal

 have h_refl_t : SPA_Deduction_Internal s Δ (ArithForm.eq t.val t.val) :=

  SPA_Deduction_Internal.eq_refl Δ t.val
```

```
 have h_weak : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq t.val t.val) := by

   apply SPA_Deduction_Internal.weaken Δ

   · exact h_refl_t

   · exact Set.subset_insert _ _


 have h_impl : SPA_Deduction_Internal s Δ (ArithForm.impl (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val)) :=

   SPA_Deduction_Internal.imp_intro Δ (ArithForm.eq c.val c.val) (ArithForm.eq t.val t.val) h_weak


 -- Clausura deductiva bajo maximalidad consistente

 exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_impl


--
============================================================================

-- 2. INFRAESTRUCTURA DE DESIGUALDAD Y REPRESENTABILIDAD DE NUMERALES

--
============================================================================


theorem prove_distinct_numerals {s : Nat} (Γ : Set (ArithForm s)) (n m : Nat) (h : n ≠ m) :

   SPA_Deduction_Internal s Γ (ArithForm.neg (ArithForm.eq (natToTerm n) (natToTerm m))) :=

 match n, m with

 | 0, 0 => (h rfl).elim

 | 0, k + 1 => by

   apply SPA_Deduction_Internal.neg_intro

   let Γ' := insert (ArithForm.eq ArithTerm.zero (ArithTerm.succ (natToTerm k))) Γ
```

```
  have h1 : SPA_Deduction_Internal s Γ' (ArithForm.eq ArithTerm.zero (ArithTerm.succ (natToTerm k))) :=
    SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
  have h2 : SPA_Deduction_Internal s Γ' (ArithForm.eq (ArithTerm.succ (natToTerm k)) ArithTerm.zero) :=
    SPA_Deduction_Internal.eq_symm _ _ _ h1
  have h3 : SPA_Deduction_Internal s Γ' (ArithForm.neg (ArithForm.eq (ArithTerm.succ (natToTerm k)) ArithTerm.zero)) := by
    apply SPA_Deduction_Internal.weaken Γ
    · exact SPA_Deduction_Internal.ax_peano1 _ _
    · exact Set.subset_insert _ _
  exact SPA_Deduction_Internal.neg_elim _ _ h2 h3
| k + 1, 0 => by
  exact SPA_Deduction_Internal.ax_peano1 _ _
| k + 1, j + 1 => by
  have h_ne : k ≠ j := fun h_eq => h (by rw [h_eq])
  apply SPA_Deduction_Internal.neg_intro
  let Γ' := insert (ArithForm.eq (ArithTerm.succ (natToTerm k)) (ArithTerm.succ (natToTerm j))) Γ
  have h1 : SPA_Deduction_Internal s Γ' (ArithForm.eq (ArithTerm.succ (natToTerm k)) (ArithTerm.succ (natToTerm j))) :=
    SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
  have h2 : SPA_Deduction_Internal s Γ' (ArithForm.impl (ArithForm.eq (ArithTerm.succ (natToTerm k)) (ArithTerm.succ (natToTerm j))) (ArithForm.eq (natToTerm k) (natToTerm j))) := by
    apply SPA_Deduction_Internal.weaken Γ
    · exact SPA_Deduction_Internal.ax_peano2 _ _ _
    · exact Set.subset_insert _ _
  have h3 : SPA_Deduction_Internal s Γ' (ArithForm.eq (natToTerm k) (natToTerm j)) :=
    SPA_Deduction_Internal.imp_elim _ _ _ h2 h1
```

```
  have h4 : SPA_Deduction_Internal s Γ' (ArithForm.neg (ArithForm.eq (natToTerm k) (natToTerm j))) := by
    apply SPA_Deduction_Internal.weaken Γ
    · exact prove_distinct_numerals Γ k j h_ne
    · exact Set.subset_insert _ _
  exact SPA_Deduction_Internal.neg_elim _ _ h3 h4

lemma prove_distinct_numerals_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) (n m : Nat) (h : n ≠ m) :
  ArithForm.neg (ArithForm.eq (natToTerm n) (natToTerm m)) ∈ Δ := by
 apply maximal_consistent_closed_under_deduction Δ h_max h_con
 exact prove_distinct_numerals Δ n m h

lemma impl_of_neg_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) (A B : ArithForm s) (h_negA : ArithForm.neg A ∈ Δ) :
  ArithForm.impl A B ∈ Δ := by
 apply maximal_consistent_closed_under_deduction Δ h_max h_con
 apply SPA_Deduction_Internal.imp_intro
 have h_A : SPA_Deduction_Internal s (insert A Δ) A :=
  SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
 have h_negA_weak : SPA_Deduction_Internal s (insert A Δ) (ArithForm.neg A) := by
  apply SPA_Deduction_Internal.weaken Δ
  · exact SPA_Deduction_Internal.hyp Δ _ h_negA
  · exact Set.subset_insert _ _
 have h_bot : SPA_Deduction_Internal s (insert A Δ) ArithForm.bottom :=
  SPA_Deduction_Internal.neg_elim _ _ h_A h_negA_weak
 exact ex_falso (insert A Δ) B h_bot
```

Lema `φ_case_correct_helper`: Transferencia de corrección local a genérica

Se demuestra un lema de transferencia fundamental que permite extender la corrección de una fórmula representadora desde una fórmula placeholder concreta φ_{place} a cualquier fórmula φ del mismo tipo estructural. Formalmente:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \forall \varphi, \varphi_{\text{place}} \in \text{ArithForm}(s+1), $$

si la instancia de la fórmula representadora construida con φ_{place} pertenece a Δ, entonces la instancia construida con φ (pero manteniendo los códigos de φ_{place} en la estructura de la implicación) también pertenece a Δ.

Estructura de la demostración

La prueba distingue dos casos según si φ coincide o no con φ_{place}:

Caso 1: $\varphi = \varphi_{\text{place}}$. La conclusión se reduce a la hipótesis `φ_place_correct`, que establece precisamente la pertenencia a Δ para la fórmula placeholder.

Caso 2: $\varphi \neq \varphi_{\text{place}}$. Este es el caso sustantivo. Se definen localmente los códigos:

- $c := \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\varphi)))$

- $t := \text{encodeForm}(\varphi)$

- $c_{\text{place}} := \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\varphi_{\text{place}})))$

- $t_{\text{place}} := \text{encodeForm}(\varphi_{\text{place}})$

La fórmula a demostrar, tras desplegar las definiciones y aplicar las sustituciones, tiene la estructura:

$$ (c = c_{\text{place}}) \to (t = t_{\text{place}}) $$

La demostración procede en tres etapas:

a) Simplificación por frescura: Se aplica `substTerm_fresh` seis veces para eliminar las sustituciones que no tienen efecto. Las justificaciones de no ocurrencia se obtienen de `NotOccur_of_isClosed` para los cuatro términos cerrados involucrados (c, t, c_{place}, t_{place}). Tras la simplificación, la fórmula se reduce a la implicación $(c = c_{\text{place}}) \to (t = t_{\text{place}})$.

b) Demostración de la distinción de códigos: Se establece que los códigos numéricos subyacentes de φ y φ_{place} son distintos. Esto se demuestra por contradicción: si $\text{encode}(\text{toSPACode}(\text{serializeForm}(\varphi))) = \text{encode}(\text{toSPACode}(\text{serializeForm}(\varphi_{\text{place}})))$, entonces por la cadena de inyectividades:

- `Encodable.encode_injective` fuerza $\text{toSPACode}(\text{serializeForm}(\varphi)) = \text{toSPACode}(\text{serializeForm}(\varphi_{\text{place}}))$

- `toSPACode_injective` fuerza $\text{serializeForm}(\varphi) = \text{serializeForm}(\varphi_{\text{place}})$

- `serializeForm_injective` fuerza $\varphi = \varphi_{\text{place}}$

lo cual contradice la hipótesis del caso. Por tanto, los códigos numéricos son distintos.

c) Aplicación del principio de explosión: Por el lema `prove_distinct_numerals_Δ`, la distinción de códigos numéricos implica que la negación de la igualdad de los numerales pertenece a Δ: $\neg(c = c_{\text{place}}) \in \Delta$. Por el lema `impl_of_neg_Δ`, la negación del antecedente fuerza que la implicación completa $(c = c_{\text{place}}) \to (t = t_{\text{place}})$ pertenece a Δ.

Intuición y necesidad

Este lema es técnicamente crucial para la estrategia de representabilidad del parser. Captura la idea de que la fórmula representadora para un caso estructural (por ejemplo, el caso `eq`) funciona correctamente para cualquier fórmula concreta de ese tipo, no solo para la fórmula placeholder utilizada en la definición. La clave lógica es que cuando la entrada no coincide con el código esperado (el de la fórmula placeholder), la implicación se satisface vacuamente porque el antecedente es falso. La distinción de códigos se garantiza por la cadena de inyectividades que recorre todos los niveles de la codificación: serialización, conversión de tipos y codificación numérica.

Lemas auxiliares de introducción disyuntiva en Δ

Se establecen versiones de las reglas de introducción de la disyunción adaptadas al contexto de una teoría maximal consistente Δ:

Lema `or_intro_left_Δ`

$$ \forall A, B \in \text{ArithForm } s,\ A \in \Delta \implies A \lor B \in \Delta $$

Lema `or_intro_right_Δ`

$$ \forall A, B \in \text{ArithForm } s,\ B \in \Delta \implies A \lor B \in \Delta $$

Intuición y necesidad

Estos lemas son adaptaciones directas de las reglas de introducción de la disyunción `or_intro_left` y `or_intro_right` demostradas anteriormente para el sistema deductivo `SPA_Deduction_Internal`. La diferencia es que aquí se trabaja directamente con la pertenencia a Δ en lugar de con la demostrabilidad. La demostración es inmediata: se toma la prueba de A (o B) desde Δ, se aplica la introducción de la disyunción correspondiente en el sistema deductivo, y se utiliza `maximal_consistent_closed_under_deduction` para transferir la demostrabilidad a pertenencia.

Estos lemas serán utilizados extensivamente en la construcción de la fórmula representadora del parser para los casos recursivos, donde será necesario combinar múltiples casos mediante disyunciones. La maximalidad de Δ garantiza que las decisiones locales sobre cada caso pueden combinarse en una fórmula disyuntiva global que cubre todas las posibilidades estructurales de una fórmula de entrada.

```
--
==========================================================================
-- 3. LEMA DE TRANSFERENCIA DE CORRECTIDAD LOCAL A GENÉRICA
--
==========================================================================

lemma φ_case_correct_helper {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)
  (φ φ_place : ArithForm (s+1))
  (φ_place_correct :
```

substForm 1 (encodeForm φ_place).val (substForm 0 (internalCode (toSPACode (serializeForm φ_place))).val (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm φ_place))).val) (ArithForm.eq (ArithTerm.var 1) (encodeForm φ_place).val))) ∈ Δ) :

substForm 1 (encodeForm φ).val (substForm 0 (internalCode (toSPACode (serializeForm φ))).val (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm φ_place))).val) (ArithForm.eq (ArithTerm.var 1) (encodeForm φ_place).val))) ∈ Δ := by

by_cases h_eq_case : φ = φ_place

· subst h_eq_case

exact φ_place_correct

· -- Introducimos localmente los let-bindings correspondientes a los términos y sus códigos

let c := internalCode (toSPACode (serializeForm φ))

let t := encodeForm φ

let c_place := internalCode (toSPACode (serializeForm φ_place))

let t_place := encodeForm φ_place

change ArithForm.impl (ArithForm.eq (substTerm 1 t.val c.val) (substTerm 1 t.val (substTerm 0 c.val c_place.val)))

(ArithForm.eq t.val (substTerm 1 t.val (substTerm 0 c.val t_place.val))) ∈ Δ

have h0 : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

have h2 : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

have h4 : NotOccur 0 c_place.val := NotOccur_of_isClosed c_place.val c_place.property 0

have h5 : NotOccur 1 c_place.val := NotOccur_of_isClosed c_place.val c_place.property 1

have h6 : NotOccur 0 t_place.val := NotOccur_of_isClosed t_place.val t_place.property 0

```
  have h7 : NotOccur 1 t_place.val := NotOccur_of_isClosed t_place.val t_place.property
1


  rw [substTerm_fresh c_place.val 0 c.val h4]

  rw [substTerm_fresh c_place.val 1 t.val h5]

  rw [substTerm_fresh t_place.val 0 c.val h6]

  rw [substTerm_fresh t_place.val 1 t.val h7]

  rw [substTerm_fresh c.val 1 t.val h2]


  have h_ne : Encodable.encode (toSPACode (serializeForm φ)) ≠ Encodable.encode
(toSPACode (serializeForm φ_place)) := by

    intro h_eq

    have h_eq' := Encodable.encode_injective h_eq

    have h_eq'' := toSPACode_injective h_eq'

    have h_eq''' := serializeForm_injective h_eq''

    exact h_eq_case h_eq'''

  have h_neg : ArithForm.neg (ArithForm.eq c.val c_place.val) ∈ Δ :=

    prove_distinct_numerals_Δ h_max h_con (Encodable.encode (toSPACode
(serializeForm φ))) (Encodable.encode (toSPACode (serializeForm φ_place))) h_ne

  exact impl_of_neg_Δ h_max h_con (ArithForm.eq c.val c_place.val) (ArithForm.eq
t.val t_place.val) h_neg


--
============================================================================

-- 4. LEMAS AUXILIARES DE INTRODUCCIÓN DISYUNTIVA EN Δ

--
============================================================================


lemma or_intro_left_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con :
Consistent Δ) (A B : ArithForm s) (hA : A ∈ Δ) :

  or_form A B ∈ Δ := by
```

```
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply or_intro_left
  exact SPA_Deduction_Internal.hyp Δ A hA

lemma or_intro_right_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) (A B : ArithForm s) (hB : B ∈ Δ) :
    or_form A B ∈ Δ := by
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply or_intro_right
  exact SPA_Deduction_Internal.hyp Δ B hB
```

Fórmula unificada `φ_parse`: La gran disyunción del parser

Se define la fórmula que unifica todos los casos del parser de fórmulas en una única disyunción de ocho ramas, cubriendo cada constructor del tipo inductivo `ArithForm`:

$$ \varphi_{\text{parse}} := \varphi_{\text{bottom}} \lor \varphi_{\text{eq}} \lor \varphi_{\text{impl}} \lor \varphi_{\text{neg}} \lor \varphi_{\forall} \lor \varphi_{\text{provable}} \lor \varphi_{\text{true}} \lor \varphi_{\text{list}} $$

Intuición y necesidad

Esta fórmula constituye la realización sintáctica completa del parser de fórmulas dentro del lenguaje objeto. Cada φ_{caso} es una fórmula con dos variables libres (v_0 y v_1) que expresa: "si v_0 es el código de una fórmula cuyo constructor principal es el correspondiente al caso, entonces v_1 es la codificación interna de dicha fórmula". La disyunción permite que el parser funcione para cualquier fórmula φ del nivel $s+1$, independientemente de su

constructor principal: al evaluar φ_{parse} con v_0 instanciado al código de φ, exactamente una de las ocho ramas tendrá el antecedente verdadero (porque el código comienza con la etiqueta numérica del constructor), y el consecuente forzará la igualdad con el código de salida.

La estructura anidada de la disyunción (asociativa a la derecha) refleja el orden de los constructores en el tipo inductivo, aunque este orden es arbitrario desde el punto de vista lógico y solo afecta a la legibilidad de las demostraciones.

Lema `φ_parse_correct`: Corrección unificada del parser

Se demuestra el teorema principal de la Fase 1.4.2.2: para cualquier fórmula φ del nivel $s+1$, la instancia de φ_{parse} con los códigos de φ pertenece a Δ:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \forall \varphi \in \text{ArithForm}(s+1),\ \varphi_{\text{parse}}[v_0 := c_\varphi,\ v_1 := t_\varphi] \in \Delta $$

donde $c_\varphi = \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\varphi)))$ y $t_\varphi = \text{encodeForm}(\varphi)$.

Estructura de la demostración

La prueba procede por casos sobre la estructura de φ (mediante `cases φ`), cubriendo exhaustivamente los ocho constructores de `ArithForm`. Para cada caso, se sigue una estrategia uniforme de construcción de la disyunción desde la hoja correspondiente hacia la raíz:

1. Caso base (`bottom`): Directamente, `φ_bottom_correct` establece que la instancia de φ_{bottom} pertenece a Δ. Por `or_intro_left_Δ`, esta pertenencia se propaga a la disyunción completa.

2. Casos intermedios (todos los demás constructores): Para un constructor genérico (por ejemplo, `eq t1 t2`), la estrategia tiene dos etapas:

- **a) Aplicación del lema de transferencia:** Se invoca `φ_case_correct_helper` con la fórmula concreta φ y la fórmula placeholder correspondiente al caso (por ejemplo, `φ_eq_correct`). Esto produce la pertenencia a Δ de la fórmula representadora del caso específico para φ, es decir, la implicación $(c_\varphi = c_{\text{place}}) \to (t_\varphi = t_{\text{place}})$.

- **b) Propagación disyuntiva:** A partir de esta pertenencia, se construye la gran disyunción de adentro hacia afuera mediante aplicaciones sucesivas de `or_intro_left_Δ` y `or_intro_right_Δ`. Por ejemplo, para el caso `eq`:

  1. Se introduce la disyunción con la rama φ_{eq} a la izquierda.

  2. Se anida esta disyunción a la derecha de φ_{bottom}.

  3. Se continúa anidando a la derecha de las disyunciones previas hasta alcanzar la raíz.

El patrón de anidamiento es estrictamente decreciente: cada paso envuelve el resultado anterior en una disyunción con la rama que está un nivel más arriba en la jerarquía de constructores.

Análisis de la estructura de propagación

La jerarquía de constructores y su orden en la disyunción es:

$$ \varphi_{\text{bottom}} \succ \varphi_{\text{eq}} \succ \varphi_{\text{impl}} \succ \varphi_{\text{neg}} \succ \varphi_{\forall} \succ \varphi_{\text{provable}} \succ \varphi_{\text{true}} \succ \varphi_{\text{list}} $$

Para un constructor en la posición k (contando desde 0), la prueba realiza exactamente $7 - k$ aplicaciones de `or_intro_right_Δ` (para envolver las ramas de la derecha) y una aplicación de `or_intro_left_Δ` (para insertar la rama correcta). Por ejemplo:

- **`bottom` (posición 0):** 0 envolturas derechas, 1 inserción izquierda.

- **`eq` (posición 1):** 1 envoltura derecha, 1 inserción izquierda.

- **`list_nth_eq` (posición 7):** 7 envolturas derechas, 1 inserción derecha.

Significado global del teorema

`φ_parse_correct` es el resultado culminante de la Fase 1.4.2.2. Establece que la fórmula φ_{parse} representa correctamente el comportamiento del parser de fórmulas para cualquier entrada. La demostración es completamente constructiva y no contiene `sorry`, lo que significa que todos los casos han sido verificados exhaustivamente.

La estructura de la prueba revela un principio de modularidad: cada caso del parser se verifica independientemente mediante su lema de corrección específico, y el lema de transferencia `φ_case_correct_helper` permite extrapolar desde la fórmula placeholder a cualquier fórmula concreta del mismo tipo estructural. La combinación de todos los casos mediante la disyunción unificada produce una fórmula que captura el comportamiento completo del parser.

Este teorema es un paso fundamental hacia la representabilidad de la relación de parseo en el sistema formal, que a su vez es un ingrediente esencial para la aritmetización de la sintaxis y los argumentos de diagonalización en el sistema SPA.

```
--
================================================================

-- 5. UNIFICACIÓN DE LA GRAN DISYUNCIÓN φ_parse Y SU CORRECTITUD (Cero sorry)

--
================================================================


def φ_parse {s : Nat} : ArithForm s :=

  or_form φ_bottom (or_form φ_eq (or_form φ_impl (or_form φ_neg

    (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))))


/--

    Lema de correctitud unificada para φ_parse (Fase 1.4.2.2 - Continuación).

    Demostrado de forma 100% constructiva y libre de `sorry` para cualquier fórmula φ genérica.

-/

lemma φ_parse_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (φ : ArithForm (s+1)) :

    let c := internalCode (toSPACode (serializeForm φ))

    let t := encodeForm φ

    substForm 1 t.val (substForm 0 c.val φ_parse) ∈ Δ := by


  intro c t

  cases φ with

  | bottom =>

    have h_bot := φ_bottom_correct Δ h_max h_con

    exact or_intro_left_Δ h_max h_con _ _ h_bot
```

```
  | eq t1 t2 =>

    have h_eq := φ_case_correct_helper Δ h_max h_con (ArithForm.eq t1 t2) (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 1) : ArithForm (s+1)) (φ_eq_correct Δ h_max h_con)

    have h_rest : or_form (substForm 1 t.val (substForm 0 c.val φ_eq)) (substForm 1 t.val (substForm 0 c.val (or_form φ_impl (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list)))))))) ∈ Δ := by

      apply or_intro_left_Δ h_max h_con

      exact h_eq

    exact or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_bottom)) _ h_rest


  | impl f1 f2 =>

    have h_impl_val := φ_case_correct_helper Δ h_max h_con (ArithForm.impl f1 f2) (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1)) (φ_impl_correct Δ h_max h_con)

    have h_rest2 : or_form (substForm 1 t.val (substForm 0 c.val φ_impl)) (substForm 1 t.val (substForm 0 c.val (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list)))))) ∈ Δ := by

      apply or_intro_left_Δ h_max h_con

      exact h_impl_val

    have h_rest1 : or_form (substForm 1 t.val (substForm 0 c.val φ_eq)) (substForm 1 t.val (substForm 0 c.val (or_form φ_impl (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))))) ∈ Δ := by

      apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_eq)) _ h_rest2

    exact or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_bottom)) _ h_rest1


  | neg f =>
```

have h_neg_val := φ_case_correct_helper Δ h_max h_con (ArithForm.neg f) (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1)) (φ_neg_correct Δ h_max h_con)

have h_rest3 : or_form (substForm 1 t.val (substForm 0 c.val φ_neg)) (substForm 1 t.val (substForm 0 c.val (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))) ∈ Δ := by

apply or_intro_left_Δ h_max h_con

exact h_neg_val

have h_rest2 : or_form (substForm 1 t.val (substForm 0 c.val φ_impl)) (substForm 1 t.val (substForm 0 c.val (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list)))))) ∈ Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_impl)) _ h_rest3

have h_rest1 : or_form (substForm 1 t.val (substForm 0 c.val φ_eq)) (substForm 1 t.val (substForm 0 c.val (or_form φ_impl (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))))) ∈ Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_eq)) _ h_rest2

exact or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_bottom)) _ h_rest1

| forall_form n f =>

have h_forall_val := φ_case_correct_helper Δ h_max h_con (ArithForm.forall_form n f) (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1)) (φ_forall_correct Δ h_max h_con)

have h_rest4 : or_form (substForm 1 t.val (substForm 0 c.val φ_forall)) (substForm 1 t.val (substForm 0 c.val (or_form φ_provable (or_form φ_true φ_list)))) ∈ Δ := by

apply or_intro_left_Δ h_max h_con

exact h_forall_val

have h_rest3 : or_form (substForm 1 t.val (substForm 0 c.val φ_neg)) (substForm 1 t.val (substForm 0 c.val (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))) ∈ Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_neg)) _ h_rest4

have h_rest2 : or_form (substForm 1 t.val (substForm 0 c.val φ_impl)) (substForm 1 t.val (substForm 0 c.val (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list)))))) ∈ Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_impl)) _ h_rest3

have h_rest1 : or_form (substForm 1 t.val (substForm 0 c.val φ_eq)) (substForm 1 t.val (substForm 0 c.val (or_form φ_impl (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list)))))))) ∈ Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_eq)) _ h_rest2

exact or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_bottom)) _ h_rest1

| provable_op h_eq term =>

have h_provable_val := φ_case_correct_helper Δ h_max h_con (ArithForm.provable_op h_eq term) (ArithForm.provable_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1)) (φ_provable_correct Δ h_max h_con)

have h_rest5 : or_form (substForm 1 t.val (substForm 0 c.val φ_provable)) (substForm 1 t.val (substForm 0 c.val (or_form φ_true φ_list))) ∈ Δ := by

apply or_intro_left_Δ h_max h_con

exact h_provable_val

have h_rest4 : or_form (substForm 1 t.val (substForm 0 c.val φ_forall)) (substForm 1 t.val (substForm 0 c.val (or_form φ_provable (or_form φ_true φ_list)))) ∈ Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_forall)) _ h_rest5

have h_rest3 : or_form (substForm 1 t.val (substForm 0 c.val φ_neg)) (substForm 1 t.val (substForm 0 c.val (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))) ∈ Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_neg)) _ h_rest4

have h_rest2 : or_form (substForm 1 t.val (substForm 0 c.val φ_impl)) (substForm 1 t.val (substForm 0 c.val (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list)))))) ∈ Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_impl)) _ h_rest3

have h_rest1 : or_form (substForm 1 t.val (substForm 0 c.val φ_eq)) (substForm 1 t.val (substForm 0 c.val (or_form φ_impl (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_eq)) _ h_rest2

exact or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_bottom)) _ h_rest1

| true_op h_eq term =>

have h_true_val := φ_case_correct_helper Δ h_max h_con (ArithForm.true_op h_eq term) (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1)) (φ_true_correct Δ h_max h_con)

have h_rest6 : or_form (substForm 1 t.val (substForm 0 c.val φ_true)) (substForm 1 t.val (substForm 0 c.val φ_list)) Δ := by

apply or_intro_left_Δ h_max h_con

exact h_true_val

have h_rest5 : or_form (substForm 1 t.val (substForm 0 c.val φ_provable)) (substForm 1 t.val (substForm 0 c.val (or_form φ_true φ_list))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_provable)) _ h_rest6

have h_rest4 : or_form (substForm 1 t.val (substForm 0 c.val φ_forall)) (substForm 1 t.val (substForm 0 c.val (or_form φ_provable (or_form φ_true φ_list)))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_forall)) _ h_rest5

have h_rest3 : or_form (substForm 1 t.val (substForm 0 c.val φ_neg)) (substForm 1 t.val (substForm 0 c.val (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_neg)) _ h_rest4

have h_rest2 : or_form (substForm 1 t.val (substForm 0 c.val φ_impl)) (substForm 1 t.val (substForm 0 c.val (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list)))))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_impl)) _ h_rest3

have h_rest1 : or_form (substForm 1 t.val (substForm 0 c.val φ_eq)) (substForm 1 t.val (substForm 0 c.val (or_form φ_impl (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_eq)) _ h_rest2

exact or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_bottom)) _ h_rest1

| list_nth_eq term1 term2 term3 =>

have h_list_val := φ_case_correct_helper Δ h_max h_con (ArithForm.list_nth_eq term1 term2 term3) (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) : ArithForm (s+1)) (φ_list_correct Δ h_max h_con)

have h_rest6 : or_form (substForm 1 t.val (substForm 0 c.val φ_true)) (substForm 1 t.val (substForm 0 c.val φ_list)) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_true)) _ h_list_val

have h_rest5 : or_form (substForm 1 t.val (substForm 0 c.val φ_provable)) (substForm 1 t.val (substForm 0 c.val (or_form φ_true φ_list))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_provable)) _ h_rest6

have h_rest4 : or_form (substForm 1 t.val (substForm 0 c.val φ_forall)) (substForm 1 t.val (substForm 0 c.val (or_form φ_provable (or_form φ_true φ_list)))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_forall)) _ h_rest5

have h_rest3 : or_form (substForm 1 t.val (substForm 0 c.val φ_neg)) (substForm 1 t.val (substForm 0 c.val (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list))))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_neg)) _ h_rest4

have h_rest2 : or_form (substForm 1 t.val (substForm 0 c.val φ_impl)) (substForm 1 t.val (substForm 0 c.val (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list)))))) Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_impl)) _ h_rest3

have h_rest1 : or_form (substForm 1 t.val (substForm 0 c.val φ_eq)) (substForm 1 t.val (substForm 0 c.val (or_form φ_impl (or_form φ_neg (or_form φ_forall (or_form φ_provable (or_form φ_true φ_list)))))))) ∈ Δ := by

apply or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_eq)) _ h_rest2

exact or_intro_right_Δ h_max h_con (substForm 1 t.val (substForm 0 c.val φ_bottom)) _ h_rest1

Fórmula `φ_bottom3`: Extensión a tres variables para el caso `bottom`

Se define una versión extendida de la fórmula representadora del parser para el caso `bottom`, incorporando una tercera variable v_2 que representa el código de la lista vacía resultante después de parsear:

$$ \varphi_{\text{bottom3}} := (v_0 = c_{\bot}) \to ((v_1 = t_{\bot}) \to (v_2 = r_{\text{nil}})) $$

donde:

- $c_{\bot} = \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\bot_{s+1})))$ es el código interno de la fórmula `bottom`,

- $t_{\bot} = \text{encodeForm}(\bot_{s+1})$ es la codificación externa de `bottom`,

- $r_{\text{nil}} = \text{internalCode}(\text{toSPACode}(\text{Code.nil}_{(s:=s+1)}))$ es el código que representa la lista vacía resultante tras consumir toda la entrada.

Intuición y necesidad

Esta fórmula extiende la representación del parser a tres variables, capturando completamente la semántica del parseo: dado un código de entrada (v_0), se produce un código de fórmula parseada (v_1) y un código de lista remanente (v_2). En el caso de `bottom`, la serialización es `[0]` (etiqueta 0 sin argumentos), por lo que tras consumir la etiqueta, la lista remanente es vacía. La fórmula expresa precisamente esta relación: si v_0 es el código de `bottom`, entonces v_1 es la codificación de `bottom` y v_2 es el código de la lista vacía. Esta versión ternaria es necesaria para la especificación completa del parser, donde no solo se relaciona entrada con salida, sino también con el resto de la entrada no consumida.

Lema `φ_bottom3_correct`: Corrección de `φ_bottom3`

Se demuestra que la instancia completa de `φ_bottom3` pertenece a cualquier teoría Δ maximalmente consistente:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \varphi_{\text{bottom3}}[v_0 := c,\ v_1 := t,\ v_2 := r] \in \Delta $$

Estructura de la demostración

A diferencia de los lemas binarios anteriores, esta prueba requiere un manejo más elaborado de las sustituciones debido a la presencia de tres variables:

1. Establecimiento de no ocurrencia: Se demuestran nueve condiciones de frescura (tres variables $\times$ tres términos cerrados). Para cada término cerrado (c, t, r) y cada variable (0, 1, 2), se invoca `NotOccur_of_isClosed`, aprovechando que los tres pertenecen al subtipo `TermClosed s`.

2. Despliegue y simplificación de sustituciones: La fórmula instanciada se expande completamente en términos de `substTerm`. Se demuestran seis igualdades auxiliares que simplifican cada una de las ocurrencias de `substTerm`:

- `h_subst0`: La sustitución triple sobre `var 0` colapsa a $c.\text{val}$. Esto se debe a que la sustitución más interna (sobre la variable 0) reemplaza `var 0` por c, y las sustituciones externas (sobre variables 1 y 2) no afectan a c por frescura.

- `h_subst1`: La sustitución triple sobre $c.\text{val}$ mismo colapsa a $c.\text{val}$, ya que ninguna variable ocurre en c.

- `h_subst2`: La sustitución triple sobre `var 1` colapsa a $t.\text{val}$, porque la sustitución sobre 1 reemplaza `var 1` por t, y la sustitución sobre 2 no afecta a t por frescura. La sustitución sobre 0 es inocua porque `var 1` no contiene a 0.

- `h_subst3`: La sustitución triple sobre $t.\text{val}$ colapsa a $t.\text{val}$.

- `h_subst4`: La sustitución triple sobre `var 2` colapsa a $r.\text{val}$.

- `h_subst5`: La sustitución triple sobre $r.\text{val}$ colapsa a $r.\text{val}$.

3. Reescritura de la fórmula: Tras aplicar las seis igualdades, la fórmula se reduce a:

$$ (c = c) \to ((t = t) \to (r = r)) $$

4. Demostración deductiva: Se procede en el sistema `SPA_Deduction_Internal`:

- Se prueba la igualdad reflexiva $r = r$.

- Se debilita al contexto con la hipótesis $(c = c)$.

- Se debilita nuevamente al contexto con ambas hipótesis $(c = c)$ y $(t = t)$.

- Se aplican dos introducciones de la implicación para obtener la fórmula completa.

5. Clausura deductiva: Se aplica `maximal_consistent_closed_under_deduction` para transferir la demostrabilidad a pertenencia en Δ.

Significado en el desarrollo del parser ternario

`φ_bottom3` y su lema de corrección establecen el patrón para la versión ternaria del parser, donde la relación de parseo es una relación entre tres códigos: entrada, fórmula parseada y lista remanente. Esta versión es necesaria para capturar completamente la semántica operacional del parser definido por `parseForm`, que devuelve un par `(fórmula, lista remanente)`. La estructura de la demostración revela que el paso de dos a tres variables no introduce complejidad conceptual adicional: la clave sigue siendo la propiedad de que los términos cerrados son inmunes a las sustituciones, lo cual permite colapsar todas las complejidades en una cadena de implicaciones entre igualdades reflexivas. Los demás casos del parser ternario seguirán este mismo patrón, con la única variación de que la tercera variable (v_2) tomará valores diferentes según la longitud del código serializado de cada constructor.

```
def φ_bottom3 {s : Nat} : ArithForm s :=
 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.bottom (s := s+1))))).val)
        (ArithForm.impl (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.bottom (s := s+1))).val)
                (ArithForm.eq (ArithTerm.var 2) (internalCode (toSPACode (_root_.Code.nil (s := s+1))))).val))
```

```
lemma φ_bottom3_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :
  let c := internalCode (toSPACode (serializeForm (ArithForm.bottom (s := s+1))))
  let t := encodeForm (ArithForm.bottom (s := s+1))
  let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))
```

```
  substForm 2 r.val (substForm 1 t.val (substForm 0 c.val φ_bottom3)) ∈ Δ := by
 intro c t r
 have h0_c : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0
 have h1_c : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1
 have h2_c : NotOccur 2 c.val := NotOccur_of_isClosed c.val c.property 2

 have h0_t : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0
 have h1_t : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1
 have h2_t : NotOccur 2 t.val := NotOccur_of_isClosed t.val t.property 2

 have h0_r : NotOccur 0 r.val := NotOccur_of_isClosed r.val r.property 0
 have h1_r : NotOccur 1 r.val := NotOccur_of_isClosed r.val r.property 1
 have h2_r : NotOccur 2 r.val := NotOccur_of_isClosed r.val r.property 2

 change ArithForm.impl
  (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))))
          (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val))))
  (ArithForm.impl
   (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 1))))
           (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val))))
   (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 2))))
           (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val))))) ∈ Δ

 have h_subst0 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))) = c.val := by
  have h_step : substTerm 0 c.val (ArithTerm.var 0) = c.val := rfl
  rw [h_step]
```

```
    rw [substTerm_fresh c.val 1 t.val h1_c]
    rw [substTerm_fresh c.val 2 r.val h2_c]

  have h_subst1 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val)) = c.val := by
    rw [substTerm_fresh c.val 0 c.val h0_c]
    rw [substTerm_fresh c.val 1 t.val h1_c]
    rw [substTerm_fresh c.val 2 r.val h2_c]

  have h_subst2 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 1))) = t.val := by
    have h_step1 : substTerm 0 c.val (ArithTerm.var 1) = ArithTerm.var 1 := rfl
    have h_step2 : substTerm 1 t.val (ArithTerm.var 1) = t.val := rfl
    rw [h_step1, h_step2]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst3 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val)) = t.val := by
    rw [substTerm_fresh t.val 0 c.val h0_t]
    rw [substTerm_fresh t.val 1 t.val h1_t]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst4 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 2))) = r.val := by
    have h_step1 : substTerm 0 c.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step2 : substTerm 1 t.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step3 : substTerm 2 r.val (ArithTerm.var 2) = r.val := rfl
    rw [h_step1, h_step2, h_step3]
```

```
  have h_subst5 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val)) = r.val
:= by
    rw [substTerm_fresh r.val 0 c.val h0_r]
    rw [substTerm_fresh r.val 1 t.val h1_r]
    rw [substTerm_fresh r.val 2 r.val h2_r]

  rw [h_subst0, h_subst1, h_subst2, h_subst3, h_subst4, h_subst5]

  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.imp_intro
  apply SPA_Deduction_Internal.imp_intro

  have h_refl : SPA_Deduction_Internal s Δ (ArithForm.eq r.val r.val) :=
SPA_Deduction_Internal.eq_refl Δ r.val
  have h_weak1 : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ)
(ArithForm.eq r.val r.val) := by
    apply SPA_Deduction_Internal.weaken Δ
    · exact h_refl
    · exact Set.subset_insert _ _
  have h_weak2 : SPA_Deduction_Internal s (insert (ArithForm.eq t.val t.val) (insert
(ArithForm.eq c.val c.val) Δ)) (ArithForm.eq r.val r.val) := by
    apply SPA_Deduction_Internal.weaken (insert (ArithForm.eq c.val c.val) Δ)
    · exact h_weak1
    · exact Set.subset_insert _ _
  exact h_weak2
```

Función `getEqParams`: Extracción de parámetros del metalenguaje

Se define una función auxiliar que extrae los códigos numéricos de los dos términos que componen una fórmula de igualdad en el nivel $s+1$. Si la fórmula es una igualdad, devuelve el par de códigos; en caso contrario, devuelve $(0, 0)$ como valor por defecto:

$$ \text{getEqParams} : \text{ArithForm}(s+1) \to \mathbb{N} \times \mathbb{N} $$

$$ \text{getEqParams}(\text{eq } t_1\ t_2) = (\text{encode}(\text{toSPACode}(\text{serializeTerm } t_1)),\ \text{encode}(\text{toSPACode}(\text{serializeTerm } t_2))) $$

$$ \text{getEqParams}(_) = (0, 0) $$

Intuición y necesidad

Esta función opera en el metalenguaje para obtener una representación numérica abstracta de los parámetros de una fórmula de igualdad. No es una función interna del lenguaje objeto, sino una herramienta del metalenguaje para construir la fórmula representadora φ_{eq3} de manera parametrizada. Al codificar los dos subtérminos con el mismo esquema utilizado en `internalCode`, se obtienen números naturales que identifican unívocamente la estructura de la igualdad. Esta parametrización permite definir una familia de fórmulas $\varphi_{\text{eq3}}(a, b)$ indexadas por los códigos de los términos, en lugar de una única fórmula placeholder con variables genéricas como en φ_{eq}.

Fórmula parametrizada `φ_eq3`: Versión ternaria para el caso `eq`

Se define la fórmula representadora del parser para el caso de igualdad, ahora parametrizada por los códigos numéricos a y b de los dos términos que la componen:

$$ \varphi_{\text{eq3}}(a, b) := (v_0 = c_{a,b}) \to ((v_1 = t_{a,b}) \to (v_2 = r_{\text{nil}})) $$

donde:

- $c_{a,b} = \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\overline{a} = \overline{b})))$ es el código interno de la fórmula de igualdad entre los numerales de Peano correspondientes a a y b,

- $t_{a,b} = \text{encodeForm}(\overline{a} = \overline{b})$ es la codificación externa de dicha fórmula,

- $r_{\text{nil}} = \text{internalCode}(\text{toSPACode}(\text{Code.nil}))$ es el código de la lista vacía.

Intuición y necesidad

Esta versión parametrizada representa un avance significativo respecto a φ_{eq}. Mientras que la versión binaria anterior utilizaba una fórmula placeholder fija con variables `var 0` y `var 1`, la versión ternaria parametrizada puede instanciarse para cualquier par concreto de códigos de términos. Esto es esencial para la representabilidad completa del parser, donde se necesita una fórmula que funcione para cualquier fórmula de igualdad, independientemente de los términos específicos que la compongan. La parametrización mediante a y b permite que la fórmula se adapte a los códigos reales de los subtérminos, en lugar de forzar una coincidencia con un placeholder fijo como en la estrategia de transferencia anterior.

Lema `φ_eq3_correct`: Corrección de `φ_eq3`

Se demuestra que para cualquier par de números naturales a y b, la instancia completa de $\varphi_{\text{eq3}}(a, b)$ pertenece a Δ:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \forall a, b \in \mathbb{N},\ \varphi_{\text{eq3}}(a, b)[v_0 := c,\ v_1 := t,\ v_2 := r] \in \Delta $$

Estructura de la demostración

La prueba es estructuralmente isomorfa a la de `φ_bottom3_correct`, reflejando la uniformidad del patrón ternario:

1. Establecimiento de no ocurrencia: Nueve condiciones de frescura para las tres variables (0, 1, 2) y los tres términos cerrados (c, t, r), todas justificadas por `NotOccur_of_isClosed`.

2. Despliegue y simplificación: La fórmula se expande completamente en términos de `substTerm`. Se demuestran seis igualdades auxiliares (h_{subst0} a h_{subst5}) que simplifican cada ocurrencia de sustitución:

- Las sustituciones sobre `var 0`, `var 1` y `var 2` colapsan respectivamente a $c.\text{val}$, $t.\text{val}$ y $r.\text{val}$, porque la sustitución relevante reemplaza la variable por el término correspondiente y las demás sustituciones son inocuas por frescura.

- Las sustituciones sobre los términos ya cerrados ($c.\text{val}$, $t.\text{val}$, $r.\text{val}$) colapsan a sí mismos por la propiedad `substTerm_fresh`, ya que ninguna variable ocurre en un término cerrado.

3. Reducción a reflexividad: Tras aplicar las seis igualdades, la fórmula se reduce a:

$$ (c = c) \to ((t = t) \to (r = r)) $$

4. Deducción: Se prueba $r = r$ por reflexividad, se debilita sucesivamente a contextos con $(c = c)$ y $(t = t)$, y se aplican dos introducciones de la implicación.

5. Clausura: Se aplica `maximal_consistent_closed_under_deduction`.

Significado de la parametrización

La parametrización de φ_{eq3} por a y b representa un cambio de estrategia respecto a los lemas anteriores. En lugar de utilizar una fórmula placeholder fija y un lema de transferencia (`φ_case_correct_helper`) para extrapolar a cualquier fórmula concreta, aquí la fórmula misma se construye en función de los parámetros reales de la igualdad. Esta aproximación es más directa y evita el razonamiento por casos sobre la distinción de códigos que requería `φ_case_correct_helper`.

La clave de esta construcción es que `getEqParams` extrae los códigos numéricos de los subtérminos en el metalenguaje, y luego `natToTerm` los reinserta como numerales en el lenguaje objeto. Esto crea un puente entre la estructura sintáctica real de la fórmula y su representación como códigos dentro del sistema formal, permitiendo que la fórmula representadora "hable" directamente sobre los códigos concretos de los subtérminos sin depender de placeholders genéricos. Este patrón de parametrización se extenderá naturalmente a los demás constructores del lenguaje.

--
==========
==

-- 1. REFACTORIZACIÓN DE LAS 7 SUBFÓRMULAS Y SUS LEMAS DE CORRECCIÓN

```
--
================================================================
==


-- ----------------------------------------------------------------

-- Caso: eq (φ_eq3 y φ_eq3_correct)

-- ----------------------------------------------------------------




--
================================================================
==

-- 1. FUNCIÓN DE EXTRACCIÓN DE PARÁMETROS DEL METALENGUAJE

--
================================================================
==


def getEqParams {s : Nat} : ArithForm (s+1) → Nat × Nat

 | ArithForm.eq t1 t2 => (Encodable.encode (toSPACode (serializeTerm t1)),
Encodable.encode (toSPACode (serializeTerm t2)))

 | _ => (0, 0)


--
================================================================
==

-- DEFINICIÓN UNIFORME Y PARAMÉTRICA DE φ_eq3 (3 VARIABLES LIBRES)

--
================================================================
==


def φ_eq3 {s : Nat} (a b : Nat) : ArithForm s :=
```

ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.eq (natToTerm (s := s+1) a) (natToTerm (s := s+1) b) : ArithForm (s+1))))).val)

(ArithForm.impl (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.eq (natToTerm (s := s+1) a) (natToTerm (s := s+1) b) : ArithForm (s+1))).val)

(ArithForm.eq (ArithTerm.var 2) (internalCode (toSPACode (_root_.Code.nil (s := s+1)))).val))

--
==========
==

-- LEMA DE CORRECCIÓN PARA φ_eq3

--
==========
==

lemma φ_eq3_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (a b : Nat) :

let c := internalCode (toSPACode (serializeForm (ArithForm.eq (natToTerm (s := s+1) a) (natToTerm (s := s+1) b) : ArithForm (s+1))))

let t := encodeForm (ArithForm.eq (natToTerm (s := s+1) a) (natToTerm (s := s+1) b) : ArithForm (s+1))

let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))

substForm 2 r.val (substForm 1 t.val (substForm 0 c.val (φ_eq3 a b))) ∈ Δ := by

intro c t r

have h0_c : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

have h1_c : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

have h2_c : NotOccur 2 c.val := NotOccur_of_isClosed c.val c.property 2

have h0_t : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

have h1_t : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1

have h2_t : NotOccur 2 t.val := NotOccur_of_isClosed t.val t.property 2

```
  have h0_r : NotOccur 0 r.val := NotOccur_of_isClosed r.val r.property 0

  have h1_r : NotOccur 1 r.val := NotOccur_of_isClosed r.val r.property 1

  have h2_r : NotOccur 2 r.val := NotOccur_of_isClosed r.val r.property 2


  change ArithForm.impl

    (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))))

              (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val))))

    (ArithForm.impl

      (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 1))))

                (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val))))

      (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 2))))

                (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val))))) ∈ Δ


  have h_subst0 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))) = c.val := by

    have h_step : substTerm 0 c.val (ArithTerm.var 0) = c.val := rfl

    rw [h_step]

    rw [substTerm_fresh c.val 1 t.val h1_c]

    rw [substTerm_fresh c.val 2 r.val h2_c]


  have h_subst1 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val)) = c.val
:= by

    rw [substTerm_fresh c.val 0 c.val h0_c]

    rw [substTerm_fresh c.val 1 t.val h1_c]

    rw [substTerm_fresh c.val 2 r.val h2_c]
```

```
  have h_subst2 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 1))) = t.val := by
    have h_step1 : substTerm 0 c.val (ArithTerm.var 1) = ArithTerm.var 1 := rfl
    have h_step2 : substTerm 1 t.val (ArithTerm.var 1) = t.val := rfl
    rw [h_step1, h_step2]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst3 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val)) = t.val := by
    rw [substTerm_fresh t.val 0 c.val h0_t]
    rw [substTerm_fresh t.val 1 t.val h1_t]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst4 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 2))) = r.val := by
    have h_step1 : substTerm 0 c.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step2 : substTerm 1 t.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step3 : substTerm 2 r.val (ArithTerm.var 2) = r.val := rfl
    rw [h_step1, h_step2, h_step3]

  have h_subst5 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val)) = r.val := by
    rw [substTerm_fresh r.val 0 c.val h0_r]
    rw [substTerm_fresh r.val 1 t.val h1_r]
    rw [substTerm_fresh r.val 2 r.val h2_r]

  rw [h_subst0, h_subst1, h_subst2, h_subst3, h_subst4, h_subst5]

  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.imp_intro
```

apply SPA_Deduction_Internal.imp_intro

have h_refl : SPA_Deduction_Internal s Δ (ArithForm.eq r.val r.val) := SPA_Deduction_Internal.eq_refl Δ r.val

have h_weak1 : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq r.val r.val) := by

apply SPA_Deduction_Internal.weaken Δ

· exact h_refl

· exact Set.subset_insert _ _

have h_weak2 : SPA_Deduction_Internal s (insert (ArithForm.eq t.val t.val) (insert (ArithForm.eq c.val c.val) Δ)) (ArithForm.eq r.val r.val) := by

apply SPA_Deduction_Internal.weaken (insert (ArithForm.eq c.val c.val) Δ)

· exact h_weak1

· exact Set.subset_insert _ _

exact h_weak2

Fórmula `φ_impl3` y lema `φ_impl3_correct`: Caso `impl` ternario

Se define la versión ternaria de la fórmula representadora para el constructor de implicación. Al igual que en el caso binario, se utiliza una fórmula placeholder fija: la implicación entre dos igualdades reflexivas con variables distinguibles:

$$ \varphi_{\text{impl3}} := (v_0 = c_{\to}) \to ((v_1 = t_{\to}) \to (v_2 = r_{\text{nil}})) $$

donde $c_{\to}$ y $t_{\to}$ son respectivamente los códigos interno y externo de la fórmula $(v_0 = v_0) \to (v_1 = v_1)$ en el nivel $s+1$.

Intuición y necesidad

Esta fórmula extiende φ_{impl} al contexto ternario, incorporando la variable v_2 para la lista remanente. La serialización de una implicación produce un código de la forma `[2, código_de_f1, código_de_f2]`. En la versión actual, se sigue utilizando un placeholder fijo, lo que significa que la tercera componente v_2 simplemente se iguala al código de la lista vacía, reflejando que tras consumir la etiqueta del constructor y los códigos de las subfórmulas, no queda resto por parsear. La demostración de corrección sigue el patrón canónico ternario ya establecido.

Fórmula `φ_neg3` y lema `φ_neg3_correct`: Caso `neg` ternario

Se define la versión ternaria para el constructor de negación, utilizando como placeholder la negación de una igualdad reflexiva:

$$ \varphi_{\text{neg3}} := (v_0 = c_{\neg}) \to ((v_1 = t_{\neg}) \to (v_2 = r_{\text{nil}})) $$

donde $c_{\neg}$ y $t_{\neg}$ son los códigos de $\neg(v_0 = v_0)$ en el nivel $s+1$.

Fórmula `φ_forall3` y lema `φ_forall3_correct`: Caso `forall_form` ternario

Se define la versión ternaria para el cuantificador universal, utilizando como placeholder $\forall v_0\ (v_0 = v_0)$:

$$ \varphi_{\forall 3} := (v_0 = c_{\forall}) \to ((v_1 = t_{\forall}) \to (v_2 = r_{\text{nil}})) $$

Uniformidad del patrón ternario

Las tres fórmulas y sus respectivos lemas de corrección comparten exactamente la misma estructura de demostración, que ya fue detallada para `φ_bottom3_correct` y `φ_eq3_correct`. El patrón consiste en:

1. **Nueve condiciones de frescura:** Para cada uno de los tres términos cerrados (c, t, r) y cada una de las tres variables (0, 1, 2), se demuestra `NotOccur` mediante `NotOccur_of_isClosed`.

2. **Seis igualdades de simplificación:** `h_subst0` a `h_subst5` reducen todas las ocurrencias de `substTerm` a los términos cerrados correspondientes, utilizando `substTerm_fresh` para eliminar las sustituciones inocuas y las reglas de sustitución básicas para las variables.

3. **Reducción a reflexividad:** La fórmula colapsa en $(c = c) \to ((t = t) \to (r = r))$.

4. **Deducción en dos pasos:** Se demuestra $r = r$ por reflexividad, se debilita dos veces añadiendo $(c = c)$ y $(t = t)$ como hipótesis, y se aplican dos introducciones de la implicación.

5. **Clausura:** `maximal_consistent_closed_under_deduction` transfiere la demostrabilidad a pertenencia en Δ.

La única variación entre los distintos casos es la fórmula concreta del nivel $s+1$ que se utiliza como placeholder, lo cual afecta únicamente a las definiciones de c y t (los códigos interno y externo). La estructura de la demostración es completamente independiente de esta elección, lo que confirma la robustez del método y su aplicabilidad a todos los constructores del lenguaje. Los casos restantes

(`provable_op`, `true_op` y `list_nth_eq`) seguirán este mismo patrón sin modificaciones sustanciales.

```
-- ---------------------------------------------------------------------

-- Caso: impl (φ_impl3 y φ_impl3_correct)

-- ---------------------------------------------------------------------


def φ_impl3 {s : Nat} : ArithForm s :=

 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1))))).val)

          (ArithForm.impl (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1))).val)

                    (ArithForm.eq (ArithTerm.var 2) (internalCode (toSPACode (_root_.Code.nil (s := s+1)))).val))


lemma φ_impl3_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

  let c := internalCode (toSPACode (serializeForm (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1))))

  let t := encodeForm (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1))

  let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))

  substForm 2 r.val (substForm 1 t.val (substForm 0 c.val φ_impl3)) ∈ Δ := by

 intro c t r
```

```
  have h0_c : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0
  have h1_c : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1
  have h2_c : NotOccur 2 c.val := NotOccur_of_isClosed c.val c.property 2

  have h0_t : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0
  have h1_t : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1
  have h2_t : NotOccur 2 t.val := NotOccur_of_isClosed t.val t.property 2

  have h0_r : NotOccur 0 r.val := NotOccur_of_isClosed r.val r.property 0
  have h1_r : NotOccur 1 r.val := NotOccur_of_isClosed r.val r.property 1
  have h2_r : NotOccur 2 r.val := NotOccur_of_isClosed r.val r.property 2

  change ArithForm.impl
    (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))))
          (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val))))
    (ArithForm.impl
     (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 1))))
           (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val))))
     (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 2))))
           (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val))))) ∈ Δ

  have h_subst0 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))) = c.val := by
    have h_step : substTerm 0 c.val (ArithTerm.var 0) = c.val := rfl
    rw [h_step]
    rw [substTerm_fresh c.val 1 t.val h1_c]
    rw [substTerm_fresh c.val 2 r.val h2_c]
```

have h_subst1 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val)) = c.val := by

rw [substTerm_fresh c.val 0 c.val h0_c]

rw [substTerm_fresh c.val 1 t.val h1_c]

rw [substTerm_fresh c.val 2 r.val h2_c]

have h_subst2 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 1))) = t.val := by

have h_step1 : substTerm 0 c.val (ArithTerm.var 1) = ArithTerm.var 1 := rfl

have h_step2 : substTerm 1 t.val (ArithTerm.var 1) = t.val := rfl

rw [h_step1, h_step2]

rw [substTerm_fresh t.val 2 r.val h2_t]

have h_subst3 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val)) = t.val := by

rw [substTerm_fresh t.val 0 c.val h0_t]

rw [substTerm_fresh t.val 1 t.val h1_t]

rw [substTerm_fresh t.val 2 r.val h2_t]

have h_subst4 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 2))) = r.val := by

have h_step1 : substTerm 0 c.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl

have h_step2 : substTerm 1 t.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl

have h_step3 : substTerm 2 r.val (ArithTerm.var 2) = r.val := rfl

rw [h_step1, h_step2, h_step3]

have h_subst5 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val)) = r.val := by

rw [substTerm_fresh r.val 0 c.val h0_r]

```
    rw [substTerm_fresh r.val 1 t.val h1_r]
    rw [substTerm_fresh r.val 2 r.val h2_r]

  rw [h_subst0, h_subst1, h_subst2, h_subst3, h_subst4, h_subst5]

  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.imp_intro
  apply SPA_Deduction_Internal.imp_intro

  have h_refl : SPA_Deduction_Internal s Δ (ArithForm.eq r.val r.val) := SPA_Deduction_Internal.eq_refl Δ r.val
  have h_weak1 : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq r.val r.val) := by
    apply SPA_Deduction_Internal.weaken Δ
    · exact h_refl
    · exact Set.subset_insert _ _
  have h_weak2 : SPA_Deduction_Internal s (insert (ArithForm.eq t.val t.val) (insert (ArithForm.eq c.val c.val) Δ)) (ArithForm.eq r.val r.val) := by
    apply SPA_Deduction_Internal.weaken (insert (ArithForm.eq c.val c.val) Δ)
    · exact h_weak1
    · exact Set.subset_insert _ _
  exact h_weak2

-- ----------------------------------------------------------------------
-- Caso: neg (φ_neg3 y φ_neg3_correct)
-- ----------------------------------------------------------------------

def φ_neg3 {s : Nat} : ArithForm s :=
```

```
 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1))))).val)

          (ArithForm.impl (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1))).val)

                    (ArithForm.eq (ArithTerm.var 2) (internalCode (toSPACode (_root_.Code.nil (s := s+1)))).val))


lemma φ_neg3_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

  let c := internalCode (toSPACode (serializeForm (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1))))

  let t := encodeForm (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1))

  let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))

  substForm 2 r.val (substForm 1 t.val (substForm 0 c.val φ_neg3)) ∈ Δ := by

 intro c t r

 have h0_c : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

 have h1_c : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

 have h2_c : NotOccur 2 c.val := NotOccur_of_isClosed c.val c.property 2


 have h0_t : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

 have h1_t : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1

 have h2_t : NotOccur 2 t.val := NotOccur_of_isClosed t.val t.property 2


 have h0_r : NotOccur 0 r.val := NotOccur_of_isClosed r.val r.property 0

 have h1_r : NotOccur 1 r.val := NotOccur_of_isClosed r.val r.property 1

 have h2_r : NotOccur 2 r.val := NotOccur_of_isClosed r.val r.property 2


 change ArithForm.impl
```

(ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 0))))

(substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val))))

(ArithForm.impl

(ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 1))))

(substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val))))

(ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 2))))

(substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val))))) ∈ Δ

have h_subst0 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 0))) = c.val := by

have h_step : substTerm 0 c.val (ArithTerm.var 0) = c.val := rfl

rw [h_step]

rw [substTerm_fresh c.val 1 t.val h1_c]

rw [substTerm_fresh c.val 2 r.val h2_c]

have h_subst1 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val)) = c.val := by

rw [substTerm_fresh c.val 0 c.val h0_c]

rw [substTerm_fresh c.val 1 t.val h1_c]

rw [substTerm_fresh c.val 2 r.val h2_c]

have h_subst2 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 1))) = t.val := by

have h_step1 : substTerm 0 c.val (ArithTerm.var 1) = ArithTerm.var 1 := rfl

have h_step2 : substTerm 1 t.val (ArithTerm.var 1) = t.val := rfl

rw [h_step1, h_step2]

rw [substTerm_fresh t.val 2 r.val h2_t]

```
  have h_subst3 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val)) = t.val
  := by

    rw [substTerm_fresh t.val 0 c.val h0_t]

    rw [substTerm_fresh t.val 1 t.val h1_t]

    rw [substTerm_fresh t.val 2 r.val h2_t]


  have h_subst4 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
  (ArithTerm.var 2))) = r.val := by

    have h_step1 : substTerm 0 c.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl

    have h_step2 : substTerm 1 t.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl

    have h_step3 : substTerm 2 r.val (ArithTerm.var 2) = r.val := rfl

    rw [h_step1, h_step2, h_step3]


  have h_subst5 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val)) = r.val
  := by

    rw [substTerm_fresh r.val 0 c.val h0_r]

    rw [substTerm_fresh r.val 1 t.val h1_r]

    rw [substTerm_fresh r.val 2 r.val h2_r]


  rw [h_subst0, h_subst1, h_subst2, h_subst3, h_subst4, h_subst5]


  apply maximal_consistent_closed_under_deduction Δ h_max h_con

  apply SPA_Deduction_Internal.imp_intro

  apply SPA_Deduction_Internal.imp_intro


  have h_refl : SPA_Deduction_Internal s Δ (ArithForm.eq r.val r.val) :=
  SPA_Deduction_Internal.eq_refl Δ r.val

  have h_weak1 : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ)
  (ArithForm.eq r.val r.val) := by
```

```
  apply SPA_Deduction_Internal.weaken Δ
  · exact h_refl
  · exact Set.subset_insert _ _
 have h_weak2 : SPA_Deduction_Internal s (insert (ArithForm.eq t.val t.val) (insert (ArithForm.eq c.val c.val) Δ)) (ArithForm.eq r.val r.val) := by
  apply SPA_Deduction_Internal.weaken (insert (ArithForm.eq c.val c.val) Δ)
  · exact h_weak1
  · exact Set.subset_insert _ _
 exact h_weak2

-- ------------------------------------------------------------------------
-- Caso: forall_form (φ_forall3 y φ_forall3_correct)
-- ------------------------------------------------------------------------

def φ_forall3 {s : Nat} : ArithForm s :=
 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1))))).val)
        (ArithForm.impl (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1))).val)
                (ArithForm.eq (ArithTerm.var 2) (internalCode (toSPACode (_root_.Code.nil (s := s+1)))).val))

lemma φ_forall3_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :
  let c := internalCode (toSPACode (serializeForm (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1))))
  let t := encodeForm (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1))
  let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))
  substForm 2 r.val (substForm 1 t.val (substForm 0 c.val φ_forall3)) ∈ Δ := by
```

```
 intro c t r
 have h0_c : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0
 have h1_c : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1
 have h2_c : NotOccur 2 c.val := NotOccur_of_isClosed c.val c.property 2

 have h0_t : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0
 have h1_t : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1
 have h2_t : NotOccur 2 t.val := NotOccur_of_isClosed t.val t.property 2

 have h0_r : NotOccur 0 r.val := NotOccur_of_isClosed r.val r.property 0
 have h1_r : NotOccur 1 r.val := NotOccur_of_isClosed r.val r.property 1
 have h2_r : NotOccur 2 r.val := NotOccur_of_isClosed r.val r.property 2

 change ArithForm.impl
   (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))))
           (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val))))
   (ArithForm.impl
    (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 1))))
           (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val))))
    (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 2))))
           (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val))))) ∈ Δ

 have h_subst0 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))) = c.val := by
   have h_step : substTerm 0 c.val (ArithTerm.var 0) = c.val := rfl
   rw [h_step]
   rw [substTerm_fresh c.val 1 t.val h1_c]
```

```
    rw [substTerm_fresh c.val 2 r.val h2_c]

  have h_subst1 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val)) = c.val
  := by
    rw [substTerm_fresh c.val 0 c.val h0_c]
    rw [substTerm_fresh c.val 1 t.val h1_c]
    rw [substTerm_fresh c.val 2 r.val h2_c]

  have h_subst2 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
  (ArithTerm.var 1))) = t.val := by
    have h_step1 : substTerm 0 c.val (ArithTerm.var 1) = ArithTerm.var 1 := rfl
    have h_step2 : substTerm 1 t.val (ArithTerm.var 1) = t.val := rfl
    rw [h_step1, h_step2]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst3 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val)) = t.val
  := by
    rw [substTerm_fresh t.val 0 c.val h0_t]
    rw [substTerm_fresh t.val 1 t.val h1_t]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst4 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
  (ArithTerm.var 2))) = r.val := by
    have h_step1 : substTerm 0 c.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step2 : substTerm 1 t.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step3 : substTerm 2 r.val (ArithTerm.var 2) = r.val := rfl
    rw [h_step1, h_step2, h_step3]

  have h_subst5 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val)) = r.val
  := by
```

```
  rw [substTerm_fresh r.val 0 c.val h0_r]
  rw [substTerm_fresh r.val 1 t.val h1_r]
  rw [substTerm_fresh r.val 2 r.val h2_r]

 rw [h_subst0, h_subst1, h_subst2, h_subst3, h_subst4, h_subst5]

 apply maximal_consistent_closed_under_deduction Δ h_max h_con
 apply SPA_Deduction_Internal.imp_intro
 apply SPA_Deduction_Internal.imp_intro

 have h_refl : SPA_Deduction_Internal s Δ (ArithForm.eq r.val r.val) := SPA_Deduction_Internal.eq_refl Δ r.val
 have h_weak1 : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq r.val r.val) := by
  apply SPA_Deduction_Internal.weaken Δ
  · exact h_refl
  · exact Set.subset_insert _ _
 have h_weak2 : SPA_Deduction_Internal s (insert (ArithForm.eq t.val t.val) (insert (ArithForm.eq c.val c.val) Δ)) (ArithForm.eq r.val r.val) := by
  apply SPA_Deduction_Internal.weaken (insert (ArithForm.eq c.val c.val) Δ)
  · exact h_weak1
  · exact Set.subset_insert _ _
 exact h_weak2
```

Fórmulas ternarias para los casos `provable_op`, `true_op` y `list_nth_eq`

Se completan las definiciones de las versiones ternarias de las fórmulas representadoras del parser para los tres constructores restantes del lenguaje `ArithForm`. Las tres construcciones siguen el patrón uniforme ya establecido.

`φ_provable3`: Caso `provable_op` ternario

$$ \varphi_{\text{provable3}} := (v_0 = c_{\text{prov}}) \to ((v_1 = t_{\text{prov}}) \to (v_2 = r_{\text{nil}})) $$

donde c_{prov} y t_{prov} son los códigos de $\text{provable_op}_{(s_{\text{lower}}:=s)}\ \text{rfl}\ (\text{var } 0)$ en el nivel $s+1$. Esta fórmula captura el caso del operador de demostrabilidad, donde la serialización tiene la etiqueta 5 seguida del código del término argumento.

`φ_true3`: Caso `true_op` ternario

$$ \varphi_{\text{true3}} := (v_0 = c_{\text{true}}) \to ((v_1 = t_{\text{true}}) \to (v_2 = r_{\text{nil}})) $$

donde c_{true} y t_{true} son los códigos de $\text{true_op}_{(s_{\text{lower}}:=s)}\ \text{rfl}\ (\text{var } 0)$. Estructuralmente idéntica al caso `provable_op`, reflejando la dualidad entre ambos operadores estratificados.

`φ_list3`: Caso `list_nth_eq` ternario

$$ \varphi_{\text{list3}} := (v_0 = c_{\text{list}}) \to ((v_1 = t_{\text{list}}) \to (v_2 = r_{\text{nil}})) $$

donde c_{list} y t_{list} son los códigos de $\text{list_nth_eq}(\text{var } 0, \text{var } 1, \text{var } 2)$. Este es el constructor de mayor aridad en el lenguaje, pero la fórmula representadora mantiene exactamente la misma estructura que los demás casos.

Lemas de corrección asociados

Cada fórmula viene acompañada de su respectivo lema de corrección: `φ_provable3_correct`, `φ_true3_correct` y `φ_list3_correct`. Las tres demostraciones son estructuralmente idénticas entre sí e idénticas a todos los casos ternarios anteriores. El patrón canónico se aplica sin variación alguna:

1. **Nueve condiciones de frescura** establecidas mediante `NotOccur_of_isClosed`.

2. **Seis igualdades de simplificación** que reducen todas las ocurrencias de `substTerm` a los términos cerrados correspondientes.

3. **Reducción a** $(c = c) \to ((t = t) \to (r = r))$.

4. **Deducción en dos pasos** con reflexividad, doble debilitamiento y doble introducción de la implicación.

5. **Clausura deductiva** mediante `maximal_consistent_closed_under_deduction`.

Significado de la completitud de los casos ternarios

Con estos tres lemas, se han cubierto exhaustivamente los ocho constructores de `ArithForm` en su versión ternaria:

| Constructor | Etiqueta | Fórmula ternaria | Lema de corrección |

|---|---|---|---|

| `bottom` | 0 | `φ_bottom3` | `φ_bottom3_correct` |

| `eq` | 1 | `φ_eq3 a b` | `φ_eq3_correct` |

| `impl` | 2 | `φ_impl3` | `φ_impl3_correct` |

| `neg` | 3 | `φ_neg3` | `φ_neg3_correct` |

| `forall_form` | 4 | `φ_forall3` | `φ_forall3_correct` |

| `provable_op` | 5 | `φ_provable3` | `φ_provable3_correct` |

| `true_op` | 6 | `φ_true3` | `φ_true3_correct` |

| `list_nth_eq` | 7 | `φ_list3` | `φ_list3_correct` |

La absoluta uniformidad de las ocho demostraciones revela un principio matemático subyacente: la corrección del parser para cualquier constructor se reduce a una tautología de la forma $(c = c) \to ((t = t) \to (r = r))$, independientemente de la complejidad del constructor. La única variación entre los casos reside en las definiciones de c y t (qué fórmula concreta del nivel $s+1$ se codifica), pero esta diferencia se resuelve completamente en el metalenguaje al momento de definir la fórmula, sin afectar la estructura de la demostración. Esta uniformidad allana el camino para la construcción de la gran disyunción ternaria `φ_parse3`, que combinará todas las ramas en una única fórmula representadora del parser completo.

-- ----------

```
-- Caso: provable_op (φ_provable3 y φ_provable3_correct)

-- ---------------------------------------------------------------------


def φ_provable3 {s : Nat} : ArithForm s :=

 ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.provable_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1))))).val)

          (ArithForm.impl (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.provable_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1))).val)

                    (ArithForm.eq (ArithTerm.var 2) (internalCode (toSPACode (_root_.Code.nil (s := s+1)))).val))


lemma φ_provable3_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

  let c := internalCode (toSPACode (serializeForm (ArithForm.provable_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1))))

  let t := encodeForm (ArithForm.provable_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1))

  let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))

  substForm 2 r.val (substForm 1 t.val (substForm 0 c.val φ_provable3)) ∈ Δ := by

 intro c t r

 have h0_c : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

 have h1_c : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

 have h2_c : NotOccur 2 c.val := NotOccur_of_isClosed c.val c.property 2


 have h0_t : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

 have h1_t : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1

 have h2_t : NotOccur 2 t.val := NotOccur_of_isClosed t.val t.property 2


 have h0_r : NotOccur 0 r.val := NotOccur_of_isClosed r.val r.property 0

 have h1_r : NotOccur 1 r.val := NotOccur_of_isClosed r.val r.property 1
```

```
  have h2_r : NotOccur 2 r.val := NotOccur_of_isClosed r.val r.property 2


  change ArithForm.impl
    (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))))
          (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val))))
    (ArithForm.impl
      (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 1))))
            (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val))))
      (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 2))))
            (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val))))) ∈ Δ


  have h_subst0 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 0))) = c.val := by
    have h_step : substTerm 0 c.val (ArithTerm.var 0) = c.val := rfl
    rw [h_step]
    rw [substTerm_fresh c.val 1 t.val h1_c]
    rw [substTerm_fresh c.val 2 r.val h2_c]


  have h_subst1 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val)) = c.val
:= by
    rw [substTerm_fresh c.val 0 c.val h0_c]
    rw [substTerm_fresh c.val 1 t.val h1_c]
    rw [substTerm_fresh c.val 2 r.val h2_c]


  have h_subst2 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 1))) = t.val := by
    have h_step1 : substTerm 0 c.val (ArithTerm.var 1) = ArithTerm.var 1 := rfl
```

```
    have h_step2 : substTerm 1 t.val (ArithTerm.var 1) = t.val := rfl
    rw [h_step1, h_step2]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst3 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val)) = t.val
:= by
    rw [substTerm_fresh t.val 0 c.val h0_t]
    rw [substTerm_fresh t.val 1 t.val h1_t]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst4 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 2))) = r.val := by
    have h_step1 : substTerm 0 c.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step2 : substTerm 1 t.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step3 : substTerm 2 r.val (ArithTerm.var 2) = r.val := rfl
    rw [h_step1, h_step2, h_step3]

  have h_subst5 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val)) = r.val
:= by
    rw [substTerm_fresh r.val 0 c.val h0_r]
    rw [substTerm_fresh r.val 1 t.val h1_r]
    rw [substTerm_fresh r.val 2 r.val h2_r]

  rw [h_subst0, h_subst1, h_subst2, h_subst3, h_subst4, h_subst5]

  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.imp_intro
  apply SPA_Deduction_Internal.imp_intro
```

have h_refl : SPA_Deduction_Internal s Δ (ArithForm.eq r.val r.val) := SPA_Deduction_Internal.eq_refl Δ r.val

have h_weak1 : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq r.val r.val) := by

apply SPA_Deduction_Internal.weaken Δ

· exact h_refl

· exact Set.subset_insert _ _

have h_weak2 : SPA_Deduction_Internal s (insert (ArithForm.eq t.val t.val) (insert (ArithForm.eq c.val c.val) Δ)) (ArithForm.eq r.val r.val) := by

apply SPA_Deduction_Internal.weaken (insert (ArithForm.eq c.val c.val) Δ)

· exact h_weak1

· exact Set.subset_insert _ _

exact h_weak2

-- ---

-- Caso: true_op (φ_true3 y φ_true3_correct)

-- ---

def φ_true3 {s : Nat} : ArithForm s :=

ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1))))).val)

(ArithForm.impl (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1))).val)

(ArithForm.eq (ArithTerm.var 2) (internalCode (toSPACode (_root_.Code.nil (s := s+1)))).val))

lemma φ_true3_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

let c := internalCode (toSPACode (serializeForm (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1))))

```
  let t := encodeForm (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1))

  let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))

  substForm 2 r.val (substForm 1 t.val (substForm 0 c.val φ_true3)) ∈ Δ := by

 intro c t r

 have h0_c : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

 have h1_c : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

 have h2_c : NotOccur 2 c.val := NotOccur_of_isClosed c.val c.property 2


 have h0_t : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

 have h1_t : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1

 have h2_t : NotOccur 2 t.val := NotOccur_of_isClosed t.val t.property 2


 have h0_r : NotOccur 0 r.val := NotOccur_of_isClosed r.val r.property 0

 have h1_r : NotOccur 1 r.val := NotOccur_of_isClosed r.val r.property 1

 have h2_r : NotOccur 2 r.val := NotOccur_of_isClosed r.val r.property 2


 change ArithForm.impl

  (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 0))))

          (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val))))

  (ArithForm.impl

   (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 1))))

           (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val))))

   (ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 2))))

           (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val))))) ∈ Δ
```

have h_subst0 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 0))) = c.val := by

have h_step : substTerm 0 c.val (ArithTerm.var 0) = c.val := rfl

rw [h_step]

rw [substTerm_fresh c.val 1 t.val h1_c]

rw [substTerm_fresh c.val 2 r.val h2_c]

have h_subst1 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val)) = c.val := by

rw [substTerm_fresh c.val 0 c.val h0_c]

rw [substTerm_fresh c.val 1 t.val h1_c]

rw [substTerm_fresh c.val 2 r.val h2_c]

have h_subst2 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 1))) = t.val := by

have h_step1 : substTerm 0 c.val (ArithTerm.var 1) = ArithTerm.var 1 := rfl

have h_step2 : substTerm 1 t.val (ArithTerm.var 1) = t.val := rfl

rw [h_step1, h_step2]

rw [substTerm_fresh t.val 2 r.val h2_t]

have h_subst3 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val)) = t.val := by

rw [substTerm_fresh t.val 0 c.val h0_t]

rw [substTerm_fresh t.val 1 t.val h1_t]

rw [substTerm_fresh t.val 2 r.val h2_t]

have h_subst4 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 2))) = r.val := by

have h_step1 : substTerm 0 c.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl

have h_step2 : substTerm 1 t.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl

```
    have h_step3 : substTerm 2 r.val (ArithTerm.var 2) = r.val := rfl
    rw [h_step1, h_step2, h_step3]

  have h_subst5 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val)) = r.val
  := by
    rw [substTerm_fresh r.val 0 c.val h0_r]
    rw [substTerm_fresh r.val 1 t.val h1_r]
    rw [substTerm_fresh r.val 2 r.val h2_r]

  rw [h_subst0, h_subst1, h_subst2, h_subst3, h_subst4, h_subst5]

  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.imp_intro
  apply SPA_Deduction_Internal.imp_intro

  have h_refl : SPA_Deduction_Internal s Δ (ArithForm.eq r.val r.val) :=
  SPA_Deduction_Internal.eq_refl Δ r.val
  have h_weak1 : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ)
  (ArithForm.eq r.val r.val) := by
    apply SPA_Deduction_Internal.weaken Δ
    · exact h_refl
    · exact Set.subset_insert _ _
  have h_weak2 : SPA_Deduction_Internal s (insert (ArithForm.eq t.val t.val) (insert
  (ArithForm.eq c.val c.val) Δ)) (ArithForm.eq r.val r.val) := by
    apply SPA_Deduction_Internal.weaken (insert (ArithForm.eq c.val c.val) Δ)
    · exact h_weak1
    · exact Set.subset_insert _ _
  exact h_weak2

-- ------------------------------------------------------------------
```

-- Caso: list_nth_eq (φ_list3 y φ_list3_correct)

-- ----------

def φ_list3 {s : Nat} : ArithForm s :=

ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode (serializeForm (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) : ArithForm (s+1))))).val)

(ArithForm.impl (ArithForm.eq (ArithTerm.var 1) (encodeForm (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) : ArithForm (s+1))).val)

(ArithForm.eq (ArithTerm.var 2) (internalCode (toSPACode (_root_.Code.nil (s := s+1)))).val))

lemma φ_list3_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) :

let c := internalCode (toSPACode (serializeForm (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) : ArithForm (s+1))))

let t := encodeForm (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) : ArithForm (s+1))

let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))

substForm 2 r.val (substForm 1 t.val (substForm 0 c.val φ_list3)) ∈ Δ := by

intro c t r

have h0_c : NotOccur 0 c.val := NotOccur_of_isClosed c.val c.property 0

have h1_c : NotOccur 1 c.val := NotOccur_of_isClosed c.val c.property 1

have h2_c : NotOccur 2 c.val := NotOccur_of_isClosed c.val c.property 2

have h0_t : NotOccur 0 t.val := NotOccur_of_isClosed t.val t.property 0

have h1_t : NotOccur 1 t.val := NotOccur_of_isClosed t.val t.property 1

have h2_t : NotOccur 2 t.val := NotOccur_of_isClosed t.val t.property 2

have h0_r : NotOccur 0 r.val := NotOccur_of_isClosed r.val r.property 0

have h1_r : NotOccur 1 r.val := NotOccur_of_isClosed r.val r.property 1

have h2_r : NotOccur 2 r.val := NotOccur_of_isClosed r.val r.property 2

change ArithForm.impl

(ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 0))))

(substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val))))

(ArithForm.impl

(ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 1))))

(substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val))))

(ArithForm.eq (substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 2))))

(substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val))))) ∈ Δ

have h_subst0 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 0))) = c.val := by

have h_step : substTerm 0 c.val (ArithTerm.var 0) = c.val := rfl

rw [h_step]

rw [substTerm_fresh c.val 1 t.val h1_c]

rw [substTerm_fresh c.val 2 r.val h2_c]

have h_subst1 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val c.val)) = c.val := by

rw [substTerm_fresh c.val 0 c.val h0_c]

rw [substTerm_fresh c.val 1 t.val h1_c]

rw [substTerm_fresh c.val 2 r.val h2_c]

have h_subst2 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val (ArithTerm.var 1))) = t.val := by

```
    have h_step1 : substTerm 0 c.val (ArithTerm.var 1) = ArithTerm.var 1 := rfl
    have h_step2 : substTerm 1 t.val (ArithTerm.var 1) = t.val := rfl
    rw [h_step1, h_step2]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst3 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val t.val)) = t.val
:= by
    rw [substTerm_fresh t.val 0 c.val h0_t]
    rw [substTerm_fresh t.val 1 t.val h1_t]
    rw [substTerm_fresh t.val 2 r.val h2_t]

  have h_subst4 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val
(ArithTerm.var 2))) = r.val := by
    have h_step1 : substTerm 0 c.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step2 : substTerm 1 t.val (ArithTerm.var 2) = ArithTerm.var 2 := rfl
    have h_step3 : substTerm 2 r.val (ArithTerm.var 2) = r.val := rfl
    rw [h_step1, h_step2, h_step3]

  have h_subst5 : substTerm 2 r.val (substTerm 1 t.val (substTerm 0 c.val r.val)) = r.val
:= by
    rw [substTerm_fresh r.val 0 c.val h0_r]
    rw [substTerm_fresh r.val 1 t.val h1_r]
    rw [substTerm_fresh r.val 2 r.val h2_r]

  rw [h_subst0, h_subst1, h_subst2, h_subst3, h_subst4, h_subst5]

  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.imp_intro
  apply SPA_Deduction_Internal.imp_intro
```

have h_refl : SPA_Deduction_Internal s Δ (ArithForm.eq r.val r.val) := SPA_Deduction_Internal.eq_refl Δ r.val

have h_weak1 : SPA_Deduction_Internal s (insert (ArithForm.eq c.val c.val) Δ) (ArithForm.eq r.val r.val) := by

apply SPA_Deduction_Internal.weaken Δ

· exact h_refl

· exact Set.subset_insert _ _

have h_weak2 : SPA_Deduction_Internal s (insert (ArithForm.eq t.val t.val) (insert (ArithForm.eq c.val c.val) Δ)) (ArithForm.eq r.val r.val) := by

apply SPA_Deduction_Internal.weaken (insert (ArithForm.eq c.val c.val) Δ)

· exact h_weak1

· exact Set.subset_insert _ _

exact h_weak2

Lema `φ_case_correct_helper3`: Transferencia de corrección local a genérica para el parser ternario

Se demuestra el análogo trivariante del lema de transferencia `φ_case_correct_helper`, extendiendo el mecanismo de extrapolación desde una fórmula placeholder φ_{place} a cualquier fórmula φ del mismo tipo estructural, ahora en el contexto del parser ternario con tres variables libres (v_0, v_1, v_2). Formalmente:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \forall \varphi, \varphi_{\text{place}} \in \text{ArithForm}(s+1), $$

si la instancia de la fórmula representadora ternaria construida con φ_{place} pertenece a Δ, entonces la instancia construida con φ (manteniendo los códigos de φ_{place} en los consecuentes de las implicaciones) también pertenece a Δ.

Estructura de la hipótesis `φ_place_correct`

La hipótesis del lema establece que la fórmula representadora para el placeholder es correcta, es decir:

$$ (v_0 = c_{\text{place}}) \to ((v_1 = t_{\text{place}}) \to (v_2 = r))[v_0 := c_{\text{place}},\ v_1 := t_{\text{place}},\ v_2 := r] \in \Delta $$

lo cual, tras las simplificaciones de sustitución, se reduce a la tautología:

$$ (c_{\text{place}} = c_{\text{place}}) \to ((t_{\text{place}} = t_{\text{place}}) \to (r = r)) \in \Delta $$

Estructura de la demostración

La prueba distingue dos casos según si φ coincide o no con φ_{place}:

Caso 1: $\varphi = \varphi_{\text{place}}$. La conclusión se reduce inmediatamente a la hipótesis `φ_place_correct`.

Caso 2: $\varphi \neq \varphi_{\text{place}}$. Se definen localmente los cinco términos cerrados involucrados:

- c', t', r': códigos de φ y la lista vacía,

- c_{place}', t_{place}': códigos de φ_{place}.

La fórmula objetivo, tras desplegar las definiciones, tiene la estructura:

$$ (c = c_{\text{place}}) \to ((t = t_{\text{place}}) \to (r = r)) $$

La demostración procede en tres etapas:

a) Simplificación por frescura: Se establecen quince condiciones de no ocurrencia (cinco términos cerrados $\times$ tres variables), todas justificadas por `NotOccur_of_isClosed`. Se aplican doce reescrituras con `substTerm_fresh` que eliminan todas las sustituciones inocuas, reduciendo cada ocurrencia de `substTerm` al término cerrado correspondiente. Tras la simplificación, la fórmula colapsa en la implicación anidada mostrada arriba.

b) Demostración de la distinción de códigos: Se establece que los códigos numéricos subyacentes de φ y φ_{place} son distintos. La cadena de inyectividades es idéntica a la del caso binario:

$$ \text{encode} \circ \text{toSPACode} \circ \text{serializeForm} $$

Si los códigos fueran iguales, la inyectividad de cada componente forzaría $\varphi = \varphi_{\text{place}}$, contradiciendo la hipótesis del caso.

c) Aplicación del principio de explosión: Por `prove_distinct_numerals_Δ`, la distinción de códigos numéricos implica $\neg(c = c_{\text{place}}) \in \Delta$. Por `impl_of_neg_Δ`, la negación del antecedente fuerza que la implicación completa $(c = c_{\text{place}}) \to ((t = t_{\text{place}}) \to (r = r))$ pertenezca a Δ.

Comparación con el caso binario

La principal diferencia técnica entre `φ_case_correct_helper3` y su versión binaria `φ_case_correct_helper` es la gestión de las condiciones de frescura. Mientras que el caso binario requería cuatro condiciones (dos términos $\times$ dos variables) y cuatro reescrituras, el caso ternario requiere quince condiciones (cinco términos $\times$ tres variables) y doce reescrituras. Sin embargo, la lógica subyacente es exactamente la misma:

- Los términos cerrados son inmunes a las sustituciones (por eso `substTerm_fresh` los colapsa).

- La distinción de códigos se hereda de la inyectividad de la cadena de codificación.

- La implicación se satisface vacuamente cuando el antecedente es falso, por el principio ex falso quodlibet.

Este lema de transferencia es la pieza clave que permitirá unificar todos los casos del parser ternario en una gran disyunción `φ_parse3`, análogamente a como `φ_case_correct_helper` permitió construir `φ_parse` en el caso binario.

--
==========
==

-- 2. REFACTORIZACIÓN COMPROMETIDA DE φ_case_correct_helper3

```
--
================================================================
==


lemma φ_case_correct_helper3 {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)
  (φ φ_place : ArithForm (s+1))
  (φ_place_correct :
   let c_place := internalCode (toSPACode (serializeForm φ_place))
   let t_place := encodeForm φ_place
   let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))
   substForm 2 r.val (substForm 1 t_place.val (substForm 0 c_place.val
    (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) c_place.val)
          (ArithForm.impl (ArithForm.eq (ArithTerm.var 1) t_place.val)
                (ArithForm.eq (ArithTerm.var 2) r.val))))) ∈ Δ) :
  let c := internalCode (toSPACode (serializeForm φ))
  let t := encodeForm φ
  let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))
  substForm 2 r.val (substForm 1 t.val (substForm 0 c.val
   (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (internalCode (toSPACode
(serializeForm φ_place))).val)
          (ArithForm.impl (ArithForm.eq (ArithTerm.var 1) (encodeForm
φ_place).val)
                (ArithForm.eq (ArithTerm.var 2) r.val))))) ∈ Δ := by
 intro c t r
 by_cases h_eq_case : φ = φ_place
 · subst h_eq_case
  exact φ_place_correct
 · let c_place := internalCode (toSPACode (serializeForm φ_place))
  let t_place := encodeForm φ_place
```

```
let c' : { x : ArithTerm s // isClosed x = true } := ⟨c.val, c.property⟩
let t' : { x : ArithTerm s // isClosed x = true } := ⟨t.val, t.property⟩
let r' : { x : ArithTerm s // isClosed x = true } := ⟨r.val, r.property⟩
let c_place' : { x : ArithTerm s // isClosed x = true } := ⟨c_place.val, c_place.property⟩
let t_place' : { x : ArithTerm s // isClosed x = true } := ⟨t_place.val, t_place.property⟩

change ArithForm.impl
 (ArithForm.eq (substTerm 2 r'.val (substTerm 1 t'.val c'.val))
         (substTerm 2 r'.val (substTerm 1 t'.val (substTerm 0 c'.val c_place'.val))))
 (ArithForm.impl
  (ArithForm.eq (substTerm 2 r'.val t'.val)
          (substTerm 2 r'.val (substTerm 1 t'.val (substTerm 0 c'.val t_place'.val))))
  (ArithForm.eq r'.val
          (substTerm 2 r'.val (substTerm 1 t'.val (substTerm 0 c'.val r'.val)))))
 ∈ Δ

have h0_c : NotOccur 0 c'.val := NotOccur_of_isClosed c'.val c'.property 0
have h1_c : NotOccur 1 c'.val := NotOccur_of_isClosed c'.val c'.property 1
have h2_c : NotOccur 2 c'.val := NotOccur_of_isClosed c'.val c'.property 2

have h0_t : NotOccur 0 t'.val := NotOccur_of_isClosed t'.val t'.property 0
have h1_t : NotOccur 1 t'.val := NotOccur_of_isClosed t'.val t'.property 1
have h2_t : NotOccur 2 t'.val := NotOccur_of_isClosed t'.val t'.property 2

have h0_r : NotOccur 0 r'.val := NotOccur_of_isClosed r'.val r'.property 0
have h1_r : NotOccur 1 r'.val := NotOccur_of_isClosed r'.val r'.property 1
have h2_r : NotOccur 2 r'.val := NotOccur_of_isClosed r'.val r'.property 2
```

```
  have h0_cp : NotOccur 0 c_place'.val := NotOccur_of_isClosed c_place'.val c_place'.property 0

  have h1_cp : NotOccur 1 c_place'.val := NotOccur_of_isClosed c_place'.val c_place'.property 1

  have h2_cp : NotOccur 2 c_place'.val := NotOccur_of_isClosed c_place'.val c_place'.property 2


  have h0_tp : NotOccur 0 t_place'.val := NotOccur_of_isClosed t_place'.val t_place'.property 0

  have h1_tp : NotOccur 1 t_place'.val := NotOccur_of_isClosed t_place'.val t_place'.property 1

  have h2_tp : NotOccur 2 t_place'.val := NotOccur_of_isClosed t_place'.val t_place'.property 2


  rw [substTerm_fresh c_place'.val 0 c'.val h0_cp]

  rw [substTerm_fresh c_place'.val 1 t'.val h1_cp]

  rw [substTerm_fresh c_place'.val 2 r'.val h2_cp]


  rw [substTerm_fresh t_place'.val 0 c'.val h0_tp]

  rw [substTerm_fresh t_place'.val 1 t'.val h1_tp]

  rw [substTerm_fresh t_place'.val 2 r'.val h2_tp]


  rw [substTerm_fresh c'.val 1 t'.val h1_c]

  rw [substTerm_fresh c'.val 2 r'.val h2_c]

  rw [substTerm_fresh t'.val 2 r'.val h2_t]


  rw [substTerm_fresh r'.val 0 c'.val h0_r]

  rw [substTerm_fresh r'.val 1 t'.val h1_r]

  rw [substTerm_fresh r'.val 2 r'.val h2_r]
```

```
  have h_ne : Encodable.encode (toSPACode (serializeForm φ)) ≠ Encodable.encode (toSPACode (serializeForm φ_place)) := by
    intro h_eq
    have h_eq' := Encodable.encode_injective h_eq
    have h_eq'' := toSPACode_injective h_eq'
    have h_eq''' := serializeForm_injective h_eq''
    exact h_eq_case h_eq'''
  have h_neg : ArithForm.neg (ArithForm.eq c'.val c_place'.val) ∈ Δ :=
    prove_distinct_numerals_Δ h_max h_con (Encodable.encode (toSPACode (serializeForm φ))) (Encodable.encode (toSPACode (serializeForm φ_place))) h_ne
  exact impl_of_neg_Δ h_max h_con (ArithForm.eq c'.val c_place'.val) (ArithForm.impl (ArithForm.eq t'.val t_place'.val) (ArithForm.eq r'.val r'.val)) h_neg
```

Fórmula unificada `φ_parse3`: La gran disyunción ternaria del parser

Se define la fórmula que unifica todos los casos del parser ternario en una única disyunción de ocho ramas, parametrizada por un par de números naturales (a, b) correspondientes a los códigos de los subtérminos en el caso de igualdad:

$$ \varphi_{\text{parse3}}(a, b) := \varphi_{\text{bottom3}} \lor \varphi_{\text{eq3}}(a, b) \lor \varphi_{\text{impl3}} \lor \varphi_{\text{neg3}} \lor \varphi_{\forall 3} \lor \varphi_{\text{provable3}} \lor \varphi_{\text{true3}} \lor \varphi_{\text{list3}} $$

Intuición y necesidad

Esta fórmula constituye la culminación de la Fase 1.4.2.2 para el parser ternario. Cada φ_{caso3} es una fórmula con tres variables libres (v_0, v_1,

v_2) que expresa la semántica completa del parseo para un constructor específico: "si v_0 es el código serializado de una fórmula cuyo constructor principal es el correspondiente al caso, entonces v_1 es la codificación interna de dicha fórmula y v_2 es el código de la lista remanente (vacía, pues se consume toda la entrada)".

La parametrización por (a, b) es necesaria únicamente para el caso `eq`, donde los códigos de los dos subtérminos deben coincidir con los de la fórmula concreta que se está parseando. Para todos los demás constructores, la parametrización es inocua (se ignora) porque sus fórmulas placeholder son fijas y el lema de transferencia `φ_case_correct_helper3` se encarga de extrapolar a cualquier fórmula concreta.

Lema `φ_parse3_correct`: Corrección unificada del parser ternario

Se demuestra el teorema principal de la representabilidad del parser ternario: para cualquier fórmula φ del nivel $s+1$, la instancia de φ_{parse3} con los parámetros extraídos de φ pertenece a Δ:

$$ \forall s \in \mathbb{N},\ \forall \Delta \subseteq \text{ArithForm } } s,\ \text{Maximal}(\Delta) \land \text{Consistent}(\Delta) \implies $$

$$ \forall \varphi \in \text{ArithForm}(s+1),\ \text{sea } } (a, b) = \text{getEqParams}(\varphi), $$

$$ \varphi_{\text{parse3}}(a, b)[v_0 := c_\varphi,\ v_1 := t_\varphi,\ v_2 := r] \in \Delta $$

donde $c_\varphi = \text{internalCode}(\text{toSPACode}(\text{serializeForm}(\varphi)))$, $t_\varphi = \text{encodeForm}(\varphi)$, y $r = \text{internalCode}(\text{toSPACode}(\text{Code.nil}))$.

Estructura de la demostración

La prueba procede por casos sobre la estructura de φ, cubriendo exhaustivamente los ocho constructores de `ArithForm`. Para cada caso, se sigue una estrategia de dos pasos:

1. Obtención de la prueba para el caso específico: Se invoca `φ_case_correct_helper3` con la fórmula concreta φ y la fórmula placeholder correspondiente a su constructor. Este lema de transferencia se apoya en el lema de corrección del caso (por ejemplo, `φ_eq3_correct` para el constructor `eq`) y produce la pertenencia a Δ de la instancia de la fórmula representadora del caso para φ.

2. Propagación disyuntiva: A partir de esta pertenencia, se construye la gran disyunción de adentro hacia afuera mediante aplicaciones sucesivas de `or_intro_left_Δ` y `or_intro_right_Δ`. El número de envolturas necesarias depende de la posición del constructor en la jerarquía:

Constructor	Posición	Envolturas derechas	Inserción
`bottom`	0	0	izquierda
`eq`	1	1	izquierda
`impl`	2	2	izquierda
`neg`	3	3	izquierda
`forall_form`	4	4	izquierda
`provable_op`	5	5	izquierda
`true_op`	6	6	izquierda
`list_nth_eq`	7	7	derecha

Para el caso `eq`, la parametrización es crucial: `getEqParams` extrae los códigos reales de los subtérminos t_1 y t_2, y la invocación a `φ_case_correct_helper3` utiliza como placeholder la igualdad entre los numerales correspondientes a esos códigos. Esto garantiza que la fórmula representadora se adapta exactamente a la estructura de la igualdad concreta.

Significado global del teorema

`φ_parse3_correct` es el resultado culminante de la aritmetización del parser de fórmulas en el sistema SPA. Establece que la relación ternaria de parseo —que asocia un código de entrada con un código de fórmula parseada y un código de lista remanente— es representable en cualquier teoría maximalmente consistente Δ.

La estructura de la demostración revela una arquitectura modular y escalable:

- **Casos base:** Cada constructor tiene su propia fórmula representadora ternaria y su lema de corrección, verificados independientemente mediante el patrón canónico de simplificación por frescura y reducción a reflexividad.

- **Transferencia:** El lema `φ_case_correct_helper3` permite extrapolar desde la fórmula placeholder a cualquier fórmula concreta del mismo tipo, utilizando la distinción de códigos y el principio de explosión para los casos no coincidentes.

- **Unificación:** La gran disyunción `φ_parse3` combina todas las ramas en una única fórmula, y la corrección se propaga mediante aplicaciones sucesivas de las reglas de introducción de la disyunción.

Este resultado es un paso fundamental hacia la demostración de los teoremas de incompletitud en el sistema SPA, ya que la representabilidad del parser es un ingrediente esencial para la construcción de la fórmula diagonal y el argumento de autorreferencia. La completitud de los ocho casos garantiza que el parser funciona correctamente para cualquier fórmula del lenguaje, cubriendo exhaustivamente toda la sintaxis de `ArithForm`.

```
--
================================================================
==

-- 4. RECOMPILACIÓN DE φ_parse3 Y φ_parse3_correct

--
================================================================
==


def φ_parse3 {s : Nat} (a b : Nat) : ArithForm s :=

  or_form φ_bottom3 (or_form (φ_eq3 a b) (or_form φ_impl3 (or_form φ_neg3

    (or_form φ_forall3 (or_form φ_provable3 (or_form φ_true3 φ_list3))))))


lemma φ_parse3_correct {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ) (φ : ArithForm (s+1)) :

    let (a, b) := getEqParams φ

    let c := internalCode (toSPACode (serializeForm φ))

    let t := encodeForm φ

    let r := internalCode (toSPACode (_root_.Code.nil (s := s+1)))

    substForm 2 r.val (substForm 1 t.val (substForm 0 c.val (φ_parse3 a b))) ∈ Δ := by

  cases φ with

  | bottom =>

    dsimp [getEqParams]

    have h_proof := φ_bottom3_correct Δ h_max h_con

    exact or_intro_left_Δ h_max h_con _ _ h_proof


  | eq t1 t2 =>

    dsimp [getEqParams]

    have h_proof := φ_case_correct_helper3 Δ h_max h_con (ArithForm.eq t1 t2)
(ArithForm.eq (natToTerm (s := s+1) (Encodable.encode (toSPACode (serializeTerm
t1)))) (natToTerm (s := s+1) (Encodable.encode (toSPACode (serializeTerm t2)))) :
ArithForm (s+1)) (φ_eq3_correct Δ h_max h_con (Encodable.encode (toSPACode
(serializeTerm t1))) (Encodable.encode (toSPACode (serializeTerm t2))))
```

```
  exact or_intro_right_Δ h_max h_con _ _
    (or_intro_left_Δ h_max h_con _ _ h_proof)

| impl f1 f2 =>
  dsimp [getEqParams]
  have h_proof := φ_case_correct_helper3 Δ h_max h_con (ArithForm.impl f1 f2) (ArithForm.impl (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) (ArithForm.eq (ArithTerm.var 1) (ArithTerm.var 1)) : ArithForm (s+1)) (φ_impl3_correct Δ h_max h_con)
  exact or_intro_right_Δ h_max h_con _ _
    (or_intro_right_Δ h_max h_con _ _
      (or_intro_left_Δ h_max h_con _ _ h_proof))

| neg f =>
  dsimp [getEqParams]
  have h_proof := φ_case_correct_helper3 Δ h_max h_con (ArithForm.neg f) (ArithForm.neg (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1)) (φ_neg3_correct Δ h_max h_con)
  exact or_intro_right_Δ h_max h_con _ _
    (or_intro_right_Δ h_max h_con _ _
      (or_intro_right_Δ h_max h_con _ _
        (or_intro_left_Δ h_max h_con _ _ h_proof)))

| forall_form n f =>
  dsimp [getEqParams]
  have h_proof := φ_case_correct_helper3 Δ h_max h_con (ArithForm.forall_form n f) (ArithForm.forall_form 0 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.var 0)) : ArithForm (s+1)) (φ_forall3_correct Δ h_max h_con)
  exact or_intro_right_Δ h_max h_con _ _
    (or_intro_right_Δ h_max h_con _ _
      (or_intro_right_Δ h_max h_con _ _
```

```
      (or_intro_right_Δ h_max h_con _ _
       (or_intro_left_Δ h_max h_con _ _ h_proof))))

 | provable_op h_op t_val =>
   dsimp [getEqParams]
   have h_proof := φ_case_correct_helper3 Δ h_max h_con (ArithForm.provable_op h_op t_val) (ArithForm.provable_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1)) (φ_provable3_correct Δ h_max h_con)
   exact or_intro_right_Δ h_max h_con _ _
     (or_intro_right_Δ h_max h_con _ _
      (or_intro_right_Δ h_max h_con _ _
       (or_intro_right_Δ h_max h_con _ _
        (or_intro_left_Δ h_max h_con _ _ h_proof))))

 | true_op h_op t_val =>
   dsimp [getEqParams]
   have h_proof := φ_case_correct_helper3 Δ h_max h_con (ArithForm.true_op h_op t_val) (ArithForm.true_op (s_lower := s) rfl (ArithTerm.var 0) : ArithForm (s+1)) (φ_true3_correct Δ h_max h_con)
   exact or_intro_right_Δ h_max h_con _ _
     (or_intro_right_Δ h_max h_con _ _
      (or_intro_right_Δ h_max h_con _ _
       (or_intro_right_Δ h_max h_con _ _
        (or_intro_right_Δ h_max h_con _ _
         (or_intro_left_Δ h_max h_con _ _ h_proof)))))

 | list_nth_eq t1 t2 t3 =>
   dsimp [getEqParams]
```

```
  have h_proof := φ_case_correct_helper3 Δ h_max h_con (ArithForm.list_nth_eq t1 t2 t3) (ArithForm.list_nth_eq (ArithTerm.var 0) (ArithTerm.var 1) (ArithTerm.var 2) : ArithForm (s+1)) (φ_list3_correct Δ h_max h_con)

  exact or_intro_right_Δ h_max h_con _ _

    (or_intro_right_Δ h_max h_con _ _

      (or_intro_right_Δ h_max h_con _ _

        (or_intro_right_Δ h_max h_con _ _

          (or_intro_right_Δ h_max h_con _ _

            (or_intro_right_Δ h_max h_con _ _

              (or_intro_right_Δ h_max h_con _ _ h_proof))))))
```

Lemas auxiliares para la manipulación de igualdades en Δ

Se establecen cuatro reglas de clausura para la relación de igualdad dentro de una teoría maximal consistente Δ. Estos lemas permiten razonar sobre igualdades como si Δ fuera cerrada bajo las reglas estándar de la lógica ecuacional.

Lema `eq_symm_Δ`: Simetría de la igualdad en Δ

$$ \forall t_1, t_2 \in \text{ArithTerm } s,\ t_1 = t_2 \in \Delta \implies t_2 = t_1 \in \Delta $$

Lema `eq_trans_Δ`: Transitividad de la igualdad en Δ

$$ \forall t_1, t_2, t_3 \in \text{ArithTerm } s,\ t_1 = t_2 \in \Delta \land t_2 = t_3 \in \Delta \implies t_1 = t_3 \in \Delta $$

Lema `eq_succ_Δ`: Congruencia del sucesor en Δ

$$ \forall t_1, t_2 \in \text{ArithTerm } s,\ t_1 = t_2 \in \Delta \implies S(t_1) = S(t_2) \in \Delta $$

Lema `eq_add_Δ`: Congruencia de la adición en Δ

$$ \forall t_1, t_2, u_1, u_2 \in \text{ArithTerm } s,\ t_1 = t_2 \in \Delta \land u_1 = u_2 \in \Delta \implies t_1 + u_1 = t_2 + u_2 \in \Delta $$

Intuición y necesidad

Estos lemas son adaptaciones directas de las reglas de inferencia del sistema deductivo `SPA_Deduction_Internal` al contexto de pertenencia a Δ. Cada demostración sigue el mismo esquema: se toma la igualdad (o igualdades) como hipótesis en Δ, se aplica la regla correspondiente en el sistema deductivo (`eq_symm`, `eq_trans`, `eq_succ`, `eq_add`), y se utiliza `maximal_consistent_closed_under_deduction` para transferir la demostrabilidad a pertenencia. La maximalidad de Δ es esencial aquí, pues garantiza que todo lo demostrable a partir de Δ está ya en Δ.

Lema `natToTerm_add_hom`: Homomorfismo de la adición

Se demuestra que la función `natToTerm` preserva la estructura de la adición: el numeral correspondiente a la suma de dos naturales es demostrablemente igual a la suma de los numerales correspondientes, dentro de cualquier teoría Δ que contenga los axiomas de la adición:

$$ \forall a, b \in \mathbb{N},\ \Delta \vdash \overline{a + b} = \overline{a} + \overline{b} $$

bajo las hipótesis de que Δ contiene todas las instancias de `add_zero_axiom` y `add_succ_axiom`.

Estructura de la demostración

La prueba procede por inducción sobre b:

Caso base ($b = 0$): $\overline{a + 0} = \overline{a}$. Por el axioma `add_zero_axiom` aplicado a $\overline{a}$, se tiene $\overline{a} + 0 = \overline{a} \in \Delta$. Por simetría de la igualdad, $\overline{a} = \overline{a} + 0 \in \Delta$, que es exactamente lo requerido ya que $\overline{a} = \text{natToTerm}(a)$ y $\overline{a} + 0 = \text{ArithTerm.add } \overline{a}\ \text{zero}$.

Paso inductivo ($b = b' + 1$): Se asume por hipótesis de inducción que $\overline{a + b'} = \overline{a} + \overline{b'} \in \Delta$. La demostración procede mediante una cadena de igualdades:

1. Por hipótesis de inducción y congruencia del sucesor: $S(\overline{a + b'}) = S(\overline{a} + \overline{b'}) \in \Delta$.

2. Por el axioma `add_succ_axiom` aplicado a $\overline{a}$ y $\overline{b'}$: $\overline{a} + S(\overline{b'}) = S(\overline{a} + \overline{b'}) \in \Delta$. Por simetría: $S(\overline{a} + \overline{b'}) = \overline{a} + S(\overline{b'}) \in \Delta$.

3. Por transitividad: $S(\overline{a + b'}) = \overline{a} + S(\overline{b'}) \in \Delta$.

4. Reconociendo que $\text{natToTerm}(a + (b' + 1)) = S(\overline{a + b'})$ y $\text{ArithTerm.add } \overline{a}\ (\text{natToTerm}(b' + 1)) = \overline{a} + S(\overline{b'})$, se obtiene exactamente la igualdad requerida.

Lema `natToTerm_mul_hom`: Homomorfismo de la multiplicación

Se demuestra el análogo para la multiplicación: el numeral del producto es demostrablemente igual al producto de los numerales:

$$ \forall a, b \in \mathbb{N},\ \Delta \vdash \overline{a \cdot b} = \overline{a} \cdot \overline{b} $$

bajo las hipótesis de que Δ contiene todas las instancias de los axiomas de la adición y la multiplicación.

Estructura de la demostración

La prueba procede por inducción sobre b:

Caso base ($b = 0$): $\overline{a \cdot 0} = 0$. Por el axioma `mul_zero_axiom` aplicado a $\overline{a}$, se tiene $\overline{a} \cdot 0 = 0 \in \Delta$. Por simetría, $0 = \overline{a} \cdot 0 \in \Delta$. Dado que $\text{natToTerm}(a \cdot 0) = \text{natToTerm}(0) = \text{zero}$, la igualdad requerida es precisamente esta.

Paso inductivo ($b = b' + 1$): La demostración utiliza una cadena de igualdades que combina la hipótesis de inducción, el homomorfismo de la adición y el axioma `mul_succ_axiom`:

1. Por el homomorfismo de la adición: $\overline{a \cdot b' + a} = \overline{a \cdot b'} + \overline{a} \in \Delta$.

2. Por hipótesis de inducción $\overline{a \cdot b'} = \overline{a} \cdot \overline{b'} \in \Delta$ y reflexividad de $\overline{a}$, se aplica congruencia de la adición: $\overline{a \cdot b'} + \overline{a} = (\overline{a} \cdot \overline{b'}) + \overline{a} \in \Delta$.

3. Por transitividad con el paso 1: $\overline{a \cdot b' + a} = (\overline{a} \cdot \overline{b'}) + \overline{a} \in \Delta$.

4. Por el axioma `mul_succ_axiom`: $\overline{a} \cdot S(\overline{b'}) = (\overline{a} \cdot \overline{b'}) + \overline{a} \in \Delta$. Por simetría: $(\overline{a} \cdot \overline{b'}) + \overline{a} = \overline{a} \cdot S(\overline{b'}) \in \Delta$.

5. Por transitividad: $\overline{a \cdot b' + a} = \overline{a} \cdot S(\overline{b'}) \in \Delta$, que es exactamente $\overline{a \cdot (b' + 1)} = \overline{a} \cdot \overline{b' + 1}$.

Significado para la aritmetización

Estos dos lemas de homomorfismo son fundamentales para la aritmetización de la sintaxis en el sistema SPA. Establecen que la representación de los números naturales como numerales de Peano se comporta correctamente respecto a las operaciones aritméticas básicas: la función `natToTerm` es un homomorfismo del semianillo $(\mathbb{N}, +, \cdot, 0, 1)$ en el semianillo de términos aritméticos módulo demostrabilidad en Δ.

Esta propiedad es indispensable para poder razonar sobre códigos dentro del lenguaje objeto. Por ejemplo, cuando se necesita expresar que el código de una fórmula compuesta se obtiene mediante operaciones aritméticas sobre los códigos de sus subfórmulas (como en la codificación de Gödel), los homomorfismos garantizan que

las manipulaciones numéricas del metalenguaje se reflejan fielmente en igualdades demostrables dentro de la teoría. La dependencia de estos lemas respecto a los axiomas aritméticos en Δ refleja el hecho de que la aritmetización de la sintaxis no es posible en una teoría puramente lógica, sino que requiere una base aritmética (los axiomas de Peano/Robinson) para establecer las propiedades básicas de los numerales.

-- Helpers para trabajar con la relación de igualdad en Δ

lemma eq_symm_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) {t1 t2 : ArithTerm s} (h : ArithForm.eq t1 t2 ∈ Δ) :

ArithForm.eq t2 t1 ∈ Δ :=

maximal_consistent_closed_under_deduction Δ h_max h_con _ (SPA_Deduction_Internal.eq_symm Δ t1 t2 (SPA_Deduction_Internal.hyp Δ _ h))

lemma eq_trans_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) {t1 t2 t3 : ArithTerm s} (h1 : ArithForm.eq t1 t2 ∈ Δ) (h2 : ArithForm.eq t2 t3 ∈ Δ) :

ArithForm.eq t1 t3 ∈ Δ :=

maximal_consistent_closed_under_deduction Δ h_max h_con _ (SPA_Deduction_Internal.eq_trans Δ t1 t2 t3 (SPA_Deduction_Internal.hyp Δ _ h1) (SPA_Deduction_Internal.hyp Δ _ h2))

lemma eq_succ_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) {t1 t2 : ArithTerm s} (h : ArithForm.eq t1 t2 ∈ Δ) :

ArithForm.eq (ArithTerm.succ t1) (ArithTerm.succ t2) ∈ Δ :=

maximal_consistent_closed_under_deduction Δ h_max h_con _ (SPA_Deduction_Internal.eq_succ Δ t1 t2 (SPA_Deduction_Internal.hyp Δ _ h))

```
lemma eq_add_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) {t1 t2 u1 u2 : ArithTerm s} (h1 : ArithForm.eq t1 t2 ∈ Δ) (h2 : ArithForm.eq u1 u2 ∈ Δ) :

  ArithForm.eq (ArithTerm.add t1 u1) (ArithTerm.add t2 u2) ∈ Δ :=

 maximal_consistent_closed_under_deduction Δ h_max h_con _ (SPA_Deduction_Internal.eq_add Δ t1 t2 u1 u2 (SPA_Deduction_Internal.hyp Δ _ h1) (SPA_Deduction_Internal.hyp Δ _ h2))


-- Homomorfismo de la adición natToTerm(a + b) = natToTerm(a) + natToTerm(b)

lemma natToTerm_add_hom {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

  (a b : Nat) :

  ArithForm.eq (natToTerm (a + b)) (ArithTerm.add (natToTerm a) (natToTerm b)) ∈ Δ := by

 induction b with

 | zero =>

  -- Base: b = 0. natToTerm (a + 0) = natToTerm a

  have h1 := h_add_zero (natToTerm a)

  exact eq_symm_Δ h_max h_con h1

 | succ b' ih =>

  -- Paso inductivo: b = b' + 1

  have h_ih_succ := eq_succ_Δ h_max h_con ih

  have h_ax := h_add_succ (natToTerm a) (natToTerm b')

  have h_ax_symm := eq_symm_Δ h_max h_con h_ax

  exact eq_trans_Δ h_max h_con h_ih_succ h_ax_symm


-- Homomorfismo de la multiplicación natToTerm(a * b) = natToTerm(a) * natToTerm(b)
```

```
lemma natToTerm_mul_hom {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)
   (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)
   (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)
   (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)
   (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)
   (a b : Nat) :
   ArithForm.eq (natToTerm (a * b)) (ArithTerm.mul (natToTerm a) (natToTerm b)) ∈
Δ := by
  induction b with
  | zero =>
   -- Base: b = 0. natToTerm (a * 0) = zero
   have h1 := h_mul_zero (natToTerm a)
   exact eq_symm_Δ h_max h_con h1
  | succ b' ih =>
   -- Paso inductivo: b = b' + 1
   have h_add := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ (a * b') a
   have h_refl := eq_refl_in_Δ Δ h_max h_con (natToTerm a)
   have h_add_ih := eq_add_Δ h_max h_con ih h_refl
   have h_trans1 := eq_trans_Δ h_max h_con h_add h_add_ih
   have h_ax := h_mul_succ (natToTerm a) (natToTerm b')
   have h_ax_symm := eq_symm_Δ h_max h_con h_ax
   exact eq_trans_Δ h_max h_con h_trans1 h_ax_symm
```

Lemas auxiliares de simplificación de sustitución

Se definen cinco lemas técnicos que simplifican la aplicación de sustituciones a fórmulas específicas que involucran igualdades y la relación de orden $<$. Todos siguen el mismo principio: cuando se sustituye una variable por un término cerrado en una fórmula, los subcomponentes cerrados de la fórmula permanecen inalterados.

Lema `substForm_eq_succ_var1`

$$ \forall d, w \in \text{TermClosed } s,\ \text{substForm}(1, w.\text{val},\ d.\text{val} = S(v_1)) = (d.\text{val} = S(w.\text{val})) $$

La sustitución de v_1 por w en la igualdad $d = S(v_1)$ solo afecta a la ocurrencia de v_1, reemplazándola por w. El término d (cerrado) no es afectado por la sustitución.

Lema `substForm_eq_zero_add_succ`

$$ \forall x, d \in \text{TermClosed } s,\ \text{substForm}(1, d.\text{val},\ 0 = x + S(v_1)) = (0 = x + S(d.\text{val})) $$

La sustitución de v_1 por d solo afecta al argumento del sucesor en la suma.

Lema `substForm_eq_succ_add_succ`

$$ \forall N \in \mathbb{N},\ \forall x, d \in \text{TermClosed } s,\ \text{substForm}(1, d.\text{val},\ S(\overline{N}) = x + S(v_1)) = (S(\overline{N}) = x + S(d.\text{val})) $$

Similar al anterior, pero con un numeral concreto $S(\overline{N})$ en el lado izquierdo. La sustitución preserva tanto el numeral como el término x (ambos cerrados).

Lema `substForm_eq_natToTerm_add_succ`

$$ \forall N \in \mathbb{N},\ \forall x, w \in \text{TermClosed } s,\ \text{substForm}(1, w.\text{val},\ \overline{N} = x + S(v_1)) = (\overline{N} = x + S(w.\text{val})) $$

Variante del lema anterior con $\overline{N}$ (posiblemente cero) en lugar de $S(\overline{N})$.

Lema `substForm_lt_form`

$$ \forall x \in \text{TermClosed } s,\ \forall M \in \text{ArithTerm } s,\ \text{isClosed}(M) = \text{true} \implies $$

$$ \text{substForm}(0, x.\text{val},\ v_0 < M) = \exists v_1\ (M = x.\text{val} + S(v_1)) $$

La sustitución de v_0 por x en la fórmula de orden estricto $v_0 < M$ (con variable testigo 1) produce la fórmula existencial que expresa que M es igual a x más algún sucesor.

Lema `zero_or_succ`: Todo numeral cerrado es cero o sucesor

Se demuestra un principio de inducción interna para términos cerrados: en cualquier teoría Δ maximal consistente con propiedad de Henkin, todo término cerrado d es demostrablemente igual a cero o a un sucesor:

$$ \forall d \in \text{TermClosed } s,\ (d = 0 \in \Delta) \lor \exists w \in \text{TermClosed } s,\ (d = S(w) \in \Delta) $$

Estructura de la demostración

La prueba es una aplicación sofisticada de los principios deductivos del sistema SPA:

1. Construcción de la fórmula de inducción: Se define $\varphi(v_0)$ como la disyunción:

$$ \varphi(v_0) := (v_0 = 0) \lor \exists v_1\ (v_0 = S(v_1)) $$

2. Caso base de la inducción: Se demuestra $\varphi(0)$, es decir, $(0 = 0) \lor \exists v_1\ (0 = S(v_1))$. La primera rama de la disyunción se satisface por reflexividad de la igualdad.

3. Paso inductivo: Se demuestra $\varphi(v_0) \to \varphi(S(v_0))$. Asumiendo $\varphi(v_0)$ (que es una disyunción), se debe probar $\varphi(S(v_0))$. La segunda rama de $\varphi(S(v_0))$ —es decir, $\exists v_1\ (S(v_0) = S(v_1))$— se satisface tomando $v_1 := v_0$, por reflexividad de la igualdad. Esto no depende de cuál rama de $\varphi(v_0)$ se cumpla.

4. Aplicación del esquema de inducción: El sistema SPA cuenta con una regla de inducción (`SPA_Deduction_Internal.induction`) que, a partir del caso base $\varphi(0)$ y del paso inductivo $\forall v_0\ (\varphi(v_0) \to \varphi(S(v_0)))$, permite concluir $\forall v_0\ \varphi(v_0)$. Esta conclusión es demostrable a partir del conjunto vacío y, por debilitamiento, a partir de Δ.

5. Instanciación al término d: Por eliminación del cuantificador universal sobre el término cerrado d, se obtiene $\varphi(d) \in \Delta$, es decir:

$$ (d = 0) \lor \exists v_1\ (d = S(v_1)) \in \Delta $$

6. Análisis por casos: Si la primera rama de la disyunción pertenece a Δ, se concluye directamente. Si no, por maximalidad de Δ, la negación de $(d = 0)$ pertenece a Δ. La estructura de la disyunción (definida como $\neg A \to B$) implica, por modus ponens con $\neg(d = 0)$, que la segunda rama $\exists v_1\ (d = S(v_1))$ pertenece a Δ. La propiedad de Henkin proporciona entonces un testigo cerrado w tal que $d = S(w) \in \Delta$.

Significado para la aritmetización

Este lema captura internamente el hecho de que todo número natural es cero o sucesor, pero aplicado a términos cerrados arbitrarios de la teoría. Es una versión sintáctica del principio de inducción que no depende de un modelo semántico, sino que se deriva puramente de las reglas deductivas del sistema SPA. La demostración ilustra la potencia del esquema de inducción interno: permite razonar por casos sobre la estructura de los términos cerrados sin necesidad de una inspección externa de su sintaxis.

Este resultado es fundamental para establecer propiedades de los códigos en el lenguaje objeto. Por ejemplo, cuando se necesita demostrar que todo código es par o impar, o que satisface alguna propiedad recursiva, `zero_or_succ` proporciona el mecanismo para descomponer el análisis en caso base y caso sucesor, facilitando las pruebas por inducción interna sobre los numerales.

```
--
================================================================

-- LEMAS AUXILIARES DE SIMPLIFICACIÓN DE SUSTITUCIÓN (SOPORTE DE TIPOS)

--
================================================================


lemma substForm_eq_succ_var1 {s : Nat} (d : { t : ArithTerm s // isClosed t = true }) (w
: { t : ArithTerm s // isClosed t = true }) :

   substForm 1 w.val (ArithForm.eq d.val (ArithTerm.succ (ArithTerm.var 1))) =

   ArithForm.eq d.val (ArithTerm.succ w.val) := by

 dsimp [substForm, substTerm]

 have h_d := substTerm_fresh d.val 1 w.val (NotOccur_of_isClosed d.val d.property 1)

 rw [h_d]


lemma substForm_eq_zero_add_succ {s : Nat} (x : { t : ArithTerm s // isClosed t = true
}) (d : { t : ArithTerm s // isClosed t = true }) :

   substForm 1 d.val (ArithForm.eq ArithTerm.zero (ArithTerm.add x.val
(ArithTerm.succ (ArithTerm.var 1)))) =

   ArithForm.eq ArithTerm.zero (ArithTerm.add x.val (ArithTerm.succ d.val)) := by

 dsimp [substForm, substTerm]

 have h_x := substTerm_fresh x.val 1 d.val (NotOccur_of_isClosed x.val x.property 1)

 rw [h_x]


lemma substForm_eq_succ_add_succ {s : Nat} (N : Nat) (x : { t : ArithTerm s // isClosed
t = true }) (d : { t : ArithTerm s // isClosed t = true }) :

   substForm 1 d.val (ArithForm.eq (ArithTerm.succ (natToTerm N)) (ArithTerm.add
x.val (ArithTerm.succ (ArithTerm.var 1)))) =

   ArithForm.eq (ArithTerm.succ (natToTerm N)) (ArithTerm.add x.val
(ArithTerm.succ d.val)) := by

 dsimp [substForm, substTerm]

 have h_N := substTerm_fresh (natToTerm N) 1 d.val (NotOccur_of_isClosed
(natToTerm N) (isClosed_natToTerm N) 1)
```

```
  have h_x := substTerm_fresh x.val 1 d.val (NotOccur_of_isClosed x.val x.property 1)
  rw [h_x, h_N]

lemma substForm_eq_natToTerm_add_succ {s : Nat} (N : Nat) (x : { t : ArithTerm s // isClosed t = true }) (w : { t : ArithTerm s // isClosed t = true }) :
    substForm 1 w.val (ArithForm.eq (natToTerm N) (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var 1)))) =
    ArithForm.eq (natToTerm N) (ArithTerm.add x.val (ArithTerm.succ w.val)) := by
  dsimp [substForm, substTerm]
  have h_N := substTerm_fresh (natToTerm N) 1 w.val (NotOccur_of_isClosed (natToTerm N) (isClosed_natToTerm N) 1)
  have h_x := substTerm_fresh x.val 1 w.val (NotOccur_of_isClosed x.val x.property 1)
  rw [h_x, h_N]

lemma substForm_lt_form {s : Nat} (x : { t : ArithTerm s // isClosed t = true }) (M_term : ArithTerm s) (h_M : isClosed M_term = true) :
    substForm 0 x.val (lt_form (ArithTerm.var 0) M_term 1) =
    exists_form 1 (ArithForm.eq M_term (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var 1)))) := by
  dsimp [lt_form, ArithForm.exists_form, exists_form, substForm, substTerm]
  have h_M_fresh := substTerm_fresh M_term 0 x.val (NotOccur_of_isClosed M_term h_M 0)
  rw [h_M_fresh]

--
============================================================

-- LEMA AUXILIAR: zero_or_succ

--
============================================================

lemma zero_or_succ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)
```

(d : TermClosed s) :

(ArithForm.eq d.val ArithTerm.zero ∈ Δ) ∨ ∃ (w : TermClosed s), ArithForm.eq d.val (ArithTerm.succ w.val) ∈ Δ := by

let φ : ArithForm s := or_form (ArithForm.eq (ArithTerm.var 0) ArithTerm.zero)

(exists_form 1 (ArithForm.eq (ArithTerm.var 0) (ArithTerm.succ (ArithTerm.var 1))))

have h_refl_zero : SPA_Deduction_Internal s (∅ : Set (ArithForm s)) (ArithForm.eq ArithTerm.zero ArithTerm.zero) :=

SPA_Deduction_Internal.eq_refl ∅ ArithTerm.zero

have h_base : SPA_Deduction_Internal s ∅ (substForm 0 ArithTerm.zero φ) :=

or_intro_left ∅ (ArithForm.eq ArithTerm.zero ArithTerm.zero) _ h_refl_zero

have h_refl_succ : SPA_Deduction_Internal s (insert φ ∅) (ArithForm.eq (ArithTerm.succ (ArithTerm.var 0)) (ArithTerm.succ (ArithTerm.var 0))) :=

SPA_Deduction_Internal.eq_refl (insert φ ∅) (ArithTerm.succ (ArithTerm.var 0))

have h_exists : SPA_Deduction_Internal s (insert φ ∅) (exists_form 1 (ArithForm.eq (ArithTerm.succ (ArithTerm.var 0)) (ArithTerm.succ (ArithTerm.var 1)))) :=

exists_intro (insert φ ∅) 1 (ArithForm.eq (ArithTerm.succ (ArithTerm.var 0)) (ArithTerm.succ (ArithTerm.var 1))) (ArithTerm.var 0) True.intro h_refl_succ

have h_step_or : SPA_Deduction_Internal s (insert φ ∅) (substForm 0 (ArithTerm.succ (ArithTerm.var 0)) φ) :=

or_intro_right (insert φ ∅) _ _ h_exists

have h_step_impl : SPA_Deduction_Internal s ∅ (ArithForm.impl φ (substForm 0 (ArithTerm.succ (ArithTerm.var 0)) φ)) :=

SPA_Deduction_Internal.imp_intro ∅ φ (substForm 0 (ArithTerm.succ (ArithTerm.var 0)) φ) h_step_or

```
 have h_step_forall : SPA_Deduction_Internal s ∅ (ArithForm.forall_form 0 (ArithForm.impl φ (substForm 0 (ArithTerm.succ (ArithTerm.var 0)) φ))) :=
  SPA_Deduction_Internal.forall_intro ∅ 0 (ArithForm.impl φ (substForm 0 (ArithTerm.succ (ArithTerm.var 0)) φ)) h_step_impl (fun ψ hψ => by nomatch hψ)

 have h_forall : SPA_Deduction_Internal s ∅ (ArithForm.forall_form 0 φ) :=
  SPA_Deduction_Internal.induction ∅ φ 0 h_base h_step_forall

 have h_forall_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form 0 φ) :=
  SPA_Deduction_Internal.weaken ∅ _ Δ h_forall (Set.empty_subset _)

 have h_free_d : IsFreeFor 0 d.val φ := IsFreeFor_closed φ 0 d
 have h_elim_d : SPA_Deduction_Internal s Δ (substForm 0 d.val φ) :=
  SPA_Deduction_Internal.forall_elim Δ 0 φ d.val h_forall_Δ h_free_d

 have h_in_Δ : substForm 0 d.val φ ∈ Δ :=
  maximal_consistent_closed_under_deduction Δ h_max h_con _ h_elim_d

 by_cases h_zero : ArithForm.eq d.val ArithTerm.zero ∈ Δ
 · left; exact h_zero
 · right
  have h_neg_zero : ArithForm.neg (ArithForm.eq d.val ArithTerm.zero) ∈ Δ := by
   cases h_max (ArithForm.eq d.val ArithTerm.zero) with
   | inl h => exact False.elim (h_zero h)
   | inr h => exact h

  have h_exists_in : exists_form 1 (ArithForm.eq d.val (ArithTerm.succ (ArithTerm.var 1))) ∈ Δ :=
   maximal_consistent_imp_closed Δ h_max h_con (ArithForm.neg (ArithForm.eq d.val ArithTerm.zero)) _ h_neg_zero h_in_Δ
```

```
  obtain ⟨w, hw⟩ := h_henkin 1 (ArithForm.eq d.val (ArithTerm.succ (ArithTerm.var 1))) h_exists_in

  have h_hw_simp := substForm_eq_succ_var1 d w

  rw [h_hw_simp] at hw

  exact ⟨w, hw⟩
```

Teorema principal `lt_numeral_bounded`: Acotación de la relación de orden para numerales

Se demuestra un resultado fundamental sobre la relación de orden estricto en el lenguaje objeto: si un término cerrado x es estrictamente menor que un numeral $\overline{M}$ según la fórmula de orden definida internamente, entonces x es demostrablemente igual a algún numeral $\overline{k}$ con $k < M$ en el sentido del metalenguaje. Formalmente:

$$ \forall M \in \mathbb{N},\ \forall x \in \text{TermClosed } s, $$

$$ \Delta \vdash x < \overline{M} \implies \exists k \in \mathbb{N},\ k < M \land \Delta \vdash x = \overline{k} $$

donde la fórmula $x < \overline{M}$ está definida mediante `lt_form` como $\exists v_1\ (\overline{M} = x + S(v_1))$, y las hipótesis sobre Δ incluyen maximalidad, consistencia, propiedad de Henkin y los axiomas de la adición.

Estructura de la demostración

La prueba procede por inducción sobre M con generalización de x:

Caso base ($M = 0$): Se asume $x < 0$, es decir, $\exists v_1\ (0 = x + S(v_1)) \in \Delta$.

1. **Extracción del testigo:** Por la propiedad de Henkin, existe un término cerrado d tal que $0 = x + S(d) \in \Delta$.

2. **Simplificación de la sustitución:** Se aplica `substForm_eq_zero_add_succ` para reescribir la fórmula instanciada como $0 = x + S(d) \in \Delta$.

3. **Aplicación del axioma de la suma:** El axioma `add_succ_axiom` para x y d establece $x + S(d) = S(x + d) \in \Delta$. Por transitividad con el paso anterior, $0 = S(x + d) \in \Delta$, y por simetría, $S(x + d) = 0 \in \Delta$.

4. **Contradicción con el primer axioma de Peano:** El axioma `ax_peano1` establece $\neg(S(y) = 0)$ para cualquier y. En particular, $\neg(S(x + d) = 0) \in \Delta$. Esto contradice la igualdad obtenida en el paso anterior, y por consistencia de Δ se obtiene una contradicción.

Conclusión del caso base: No puede existir $x < 0$, lo cual es coherente con la interpretación pretendida.

Paso inductivo ($M = N + 1$): Se asume $x < \overline{N + 1}$, es decir, $\exists v_1\ (S(\overline{N}) = x + S(v_1)) \in \Delta$.

1. **Extracción del testigo:** Por Henkin, existe d cerrado tal que $S(\overline{N}) = x + S(d) \in \Delta$.

2. **Simplificación y uso del axioma de la suma:** Se aplica `substForm_eq_succ_add_succ` y el axioma `add_succ_axiom` para obtener $S(\overline{N}) = S(x + d) \in \Delta$.

3. **Aplicación del segundo axioma de Peano:** El axioma `ax_peano2` (inyectividad del sucesor) implica $S(\overline{N}) = S(x + d) \to \overline{N} = x + d \in \Delta$. Por modus ponens con el paso anterior, $\overline{N} = x + d \in \Delta$.

4. **Análisis por casos sobre d:** Se aplica el lema `zero_or_succ` al término d:

- **Caso $d = 0$:** Entonces $\overline{N} = x + 0 \in \Delta$. Por el axioma `add_zero_axiom`, $x + 0 = x \in \Delta$. Por transitividad, $\overline{N} = x \in \Delta$, y por simetría, $x = \overline{N} \in \Delta$. Se elige $k = N$, que cumple $k < N + 1$ y $x = \overline{k} \in \Delta$.

- **Caso $d = S(w)$ para algún w:** Entonces $\overline{N} = x + S(w) \in \Delta$. Esta igualdad puede reescribirse, mediante `substForm_eq_natToTerm_add_succ`, como la instancia de la fórmula $\psi(v_1) := (\overline{N} = x + S(v_1))$ con $v_1 := w$. Aplicando introducción del existencial, se obtiene $\exists v_1\ (\overline{N} = x + S(v_1)) \in \Delta$, que es precisamente $x < \overline{N}$.

5. **Aplicación de la hipótesis de inducción:** Dado que ahora se tiene $x < \overline{N} \in \Delta$, la hipótesis de inducción (aplicada a N y al mismo x) proporciona un $k < N$ tal que $x = \overline{k} \in \Delta$. Como $k < N$, se tiene $k < N + 1$, completando la demostración.

Significado para la aritmetización de la sintaxis

El teorema `lt_numeral_bounded` es un resultado de adecuación entre la noción interna de orden (definida sintácticamente mediante `lt_form`) y la noción externa de orden en el metalenguaje. Establece que la fórmula $x < \overline{M}$ captura correctamente la propiedad de que x representa un número natural estrictamente menor que M, en el sentido de que x es demostrablemente igual a un numeral $\overline{k}$ con $k < M$.

Este resultado tiene consecuencias profundas para la Gödelización:

1. **Acotación de códigos:** Cuando se codifican objetos sintácticos, los códigos resultantes son números naturales concretos. El teorema permite demostrar que si un código satisface internamente una desigualdad con un numeral, entonces externamente es menor que ese numeral, lo cual es indispensable para establecer cotas en las definiciones recursivas.

2. **Principio del mínimo:** El teorema implica que el orden interno sobre numerales es bien fundado en el sentido de que no existen cadenas descendentes infinitas demostrables, una propiedad necesaria para la consistencia de los argumentos inductivos internos.

3. **Corrección de la función beta:** En la representación de secuencias finitas mediante la función beta de Gödel, es necesario acotar los elementos de la secuencia. Este teorema proporciona la herramienta para traducir cotas internas (expresadas con `lt_form`) en cotas externas (desigualdades en $\mathbb{N}$), cerrando el círculo entre la aritmética formal y el metalenguaje.

--
==

-- TEOREMA PRINCIPAL: lt_numeral_bounded

```
--
============================================================


lemma lt_numeral_bounded {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

  (M : Nat) (x : TermClosed s)

  (h_lt : substForm 0 x.val (lt_form (ArithTerm.var 0) (natToTerm M) 1) ∈ Δ) :

  ∃ (k : Nat), k < M ∧ ArithForm.eq x.val (natToTerm k) ∈ Δ := by

 induction M generalizing x with

 | zero =>

  have h_lt_simp := substForm_lt_form x (natToTerm 0) (isClosed_natToTerm 0)

  rw [h_lt_simp] at h_lt


  have h_exists_in : exists_form 1 (ArithForm.eq (natToTerm 0) (ArithTerm.add x.val
(ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ := h_lt

  obtain ⟨d, hd⟩ := h_henkin 1 (ArithForm.eq (natToTerm 0) (ArithTerm.add x.val
(ArithTerm.succ (ArithTerm.var 1)))) h_exists_in


  change substForm 1 d.val (ArithForm.eq ArithTerm.zero (ArithTerm.add x.val
(ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ at hd

  have h_hd_simp := substForm_eq_zero_add_succ x d

  rw [h_hd_simp] at hd


  have hd_in : ArithForm.eq ArithTerm.zero (ArithTerm.add x.val (ArithTerm.succ
d.val)) ∈ Δ := hd

  have h_ax := h_add_succ x.val d.val


  have h_trans := eq_trans_Δ h_max h_con hd_in h_ax
```

```
  have h_trans_symm := eq_symm_Δ h_max h_con h_trans

  have h_peano1 := SPA_Deduction_Internal.ax_peano1 Δ (ArithTerm.add x.val d.val)
  have h_peano1_in : ArithForm.neg (ArithForm.eq (ArithTerm.succ (ArithTerm.add x.val d.val)) ArithTerm.zero) ∈ Δ :=
    maximal_consistent_closed_under_deduction Δ h_max h_con _ h_peano1

  have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
   SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_trans_symm) (SPA_Deduction_Internal.hyp Δ _ h_peano1_in)
  exact False.elim (h_con h_bot)

 | succ N ih =>
  have h_lt_simp := substForm_lt_form x (natToTerm (N + 1)) (isClosed_natToTerm (N + 1))
  rw [h_lt_simp] at h_lt

  have h_exists_in : exists_form 1 (ArithForm.eq (ArithTerm.succ (natToTerm N)) (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ := h_lt
  obtain ⟨d, hd⟩ := h_henkin 1 (ArithForm.eq (ArithTerm.succ (natToTerm N)) (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var 1)))) h_exists_in

  have h_hd_simp := substForm_eq_succ_add_succ N x d
  rw [h_hd_simp] at hd

  have h_ax := h_add_succ x.val d.val
  have h_trans := eq_trans_Δ h_max h_con hd h_ax

  have h_peano2 := SPA_Deduction_Internal.ax_peano2 Δ (natToTerm N) (ArithTerm.add x.val d.val)
```

```
  have h_peano2_in : ArithForm.impl (ArithForm.eq (ArithTerm.succ (natToTerm N)) (ArithTerm.succ (ArithTerm.add x.val d.val))) (ArithForm.eq (natToTerm N) (ArithTerm.add x.val d.val)) ∈ Δ :=
    maximal_consistent_closed_under_deduction Δ h_max h_con _ h_peano2

  have h_eq_N : ArithForm.eq (natToTerm N) (ArithTerm.add x.val d.val) ∈ Δ :=
    maximal_consistent_imp_closed Δ h_max h_con _ _ h_trans h_peano2_in

  cases zero_or_succ Δ h_max h_con h_henkin d with
  | inl h_d_zero =>
    have h_add_z := h_add_zero x.val
    have h_refl_x := eq_refl_in_Δ Δ h_max h_con x.val
    have h_congr := eq_add_Δ h_max h_con h_refl_x h_d_zero
    have h_trans2 := eq_trans_Δ h_max h_con h_congr h_add_z
    have h_final := eq_trans_Δ h_max h_con h_eq_N h_trans2
    have h_final_symm := eq_symm_Δ h_max h_con h_final

    use N
    refine ⟨by omega, h_final_symm⟩

  | inr h_d_succ =>
    obtain ⟨w, hw⟩ := h_d_succ
    have h_refl_x := eq_refl_in_Δ Δ h_max h_con x.val
    have h_congr := eq_add_Δ h_max h_con h_refl_x hw
    have h_trans2 := eq_trans_Δ h_max h_con h_eq_N h_congr

    let ψ_lt := ArithForm.eq (natToTerm N) (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var 1)))
```

```
    have h_trans2_simp := substForm_eq_natToTerm_add_succ N x w
    have h_trans2_rew : substForm 1 w.val ψ_lt ∈ Δ := by
      rw [h_trans2_simp]
      exact h_trans2

    have h_free : IsFreeFor 1 w.val ψ_lt := IsFreeFor_closed ψ_lt 1 w
    have h_exists_ded : SPA_Deduction_Internal s Δ (exists_form 1 ψ_lt) :=
      exists_intro Δ 1 ψ_lt w.val h_free (SPA_Deduction_Internal.hyp Δ _ h_trans2_rew)

    have h_exists_in' : exists_form 1 ψ_lt ∈ Δ :=
      maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists_ded

    have h_lt_simp_N := substForm_lt_form x (natToTerm N) (isClosed_natToTerm N)
    have h_ih_lt : substForm 0 x.val (lt_form (ArithTerm.var 0) (natToTerm N) 1) ∈ Δ :=
by
      rw [h_lt_simp_N]
      exact h_exists_in'

    obtain ⟨k, hk⟩ := ih x h_ih_lt

    use k
    refine ⟨by omega, hk.2⟩
```

Lemas lógicos y auxiliares de soporte para Δ

Se establece una batería de lemas que extienden el repertorio de reglas de inferencia disponibles para razonar sobre la pertenencia de fórmulas a una teoría maximal consistente Δ. Estos lemas cubren la eliminación de la conjunción, la resolución de disyunciones, y la introducción de negaciones de conjunciones.

Lema `and_elim_left`: Eliminación izquierda de la conjunción en el sistema deductivo

$$ \forall \Gamma \subseteq \text{ArithForm } s,\ \forall A, B \in \text{ArithForm } s,\ \Gamma \vdash A \land B \implies \Gamma \vdash A $$

Intuición y necesidad

La conjunción $A \land B$ se definió como $\neg(A \to \neg B)$. La demostración de esta regla de eliminación es más compleja que su análoga en deducción natural debido a la definición negativa. La estrategia consiste en:

1. Asumir $\neg A$ en el contexto y demostrar $A \to \neg B$ (usando ex falso a partir de A y $\neg A$).

2. La hipótesis principal $\neg(A \to \neg B)$ (que es $A \land B$) se debilita al contexto extendido.

3. Se aplica eliminación de la negación para obtener $\bot$, y luego doble negación para obtener A.

Lema `and_elim_left_Δ` y `and_elim_right_Δ`: Eliminación de la conjunción en Δ

$$ A \land B \in \Delta \implies A \in \Delta $$

$$ A \land B \in \Delta \implies B \in \Delta $$

Ambos lemas transfieren las reglas de eliminación de la conjunción al contexto de pertenencia a Δ, utilizando `maximal_consistent_closed_under_deduction` para cerrar la deducción.

Lema `or_resolve_left`: Resolución de disyunción por negación del segundo disyunto

$$ A \lor B \in \Delta \land \neg B \in \Delta \implies A \in \Delta $$

Intuición y necesidad

Esta regla captura el silogismo disyuntivo: si se sabe que $A \lor B$ y además $\neg B$, entonces debe ser A. La demostración utiliza la maximalidad de Δ: si $A \notin \Delta$, entonces por maximalidad $\neg A \in \Delta$. Dado que $A \lor B$ se define como $\neg A \to B$, y se tiene $\neg A \in \Delta$, se deduce $B \in \Delta$. Pero también se tiene $\neg B \in \Delta$, lo que contradice la consistencia. Por tanto, $A \in \Delta$.

Lema `neg_and_intro_left`: Introducción de negación de conjunción por negación del primer conjuncto

$$ \neg A \in \Delta \implies \neg(A \land B) \in \Delta $$

Lema `neg_and_intro_right`: Introducción de negación de conjunción por negación del segundo conjuncto

$$ \neg B \in \Delta \implies \neg(A \land B) \in \Delta $$

Intuición y necesidad

Estos lemas capturan las leyes de De Morgan para la conjunción en el contexto de Δ: si uno de los conjunctos es falso, la conjunción completa es falsa. La demostración del primer lema utiliza `impl_of_neg_Δ` para obtener $A \to \neg B \in \Delta$ a partir de $\neg A \in \Delta$. Luego, dado que $A \land B$ es por definición $\neg(A \to \neg B)$, se introduce la negación para obtener $\neg(A \land B)$. El segundo lema es análogo pero más directo: $\neg B \in \Delta$ implica $A \to \neg B \in \Delta$ por introducción de la implicación con debilitamiento, y se procede como en el caso anterior.

Lema `neg_neg_intro_Δ`: Doble negación en Δ

$$ A \in \Delta \implies \neg\neg A \in \Delta $$

La demostración es directa: se aplica introducción de la negación en el sistema deductivo, asumiendo $\neg A$ y derivando una contradicción con A (que está en Δ y se debilita al contexto extendido).

Significado para la infraestructura deductiva

Estos lemas completan el conjunto de herramientas necesarias para razonar sobre fórmulas dentro de una teoría maximal consistente Δ sin tener que invocar constantemente el sistema deductivo subyacente. Proporcionan una interfaz de alto

nivel donde las reglas lógicas estándar (eliminación de conjunción, silogismo disyuntivo, leyes de De Morgan, doble negación) pueden aplicarse directamente a la pertenencia en Δ, ocultando la complejidad de las definiciones de los conectivos en términos de implicación y negación. Esta infraestructura será indispensable para las demostraciones de representabilidad más complejas, donde se necesita manipular fórmulas compuestas dentro de Δ de manera ágil y modular.

```
--
============================================================

-- LEMAS LÓGICOS Y AUXILIARES DE SOPORTE

--
============================================================


lemma and_elim_left {s : Nat} (Γ : Set (ArithForm s)) (A B : ArithForm s)

    (h : SPA_Deduction_Internal s Γ (and_form A B)) :

    SPA_Deduction_Internal s Γ A := by

  apply SPA_Deduction_Internal.double_neg

  apply SPA_Deduction_Internal.neg_intro

  have h_imp : SPA_Deduction_Internal s (insert (ArithForm.neg A) Γ) (ArithForm.impl
A (ArithForm.neg B)) := by

    apply SPA_Deduction_Internal.imp_intro

    have h_A : SPA_Deduction_Internal s (insert A (insert (ArithForm.neg A) Γ)) A :=

      SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)

    have h_negA : SPA_Deduction_Internal s (insert A (insert (ArithForm.neg A) Γ))
(ArithForm.neg A) :=
```

```
    SPA_Deduction_Internal.hyp _ _ (Or.inr (Or.inl rfl))
  have h_bot := SPA_Deduction_Internal.neg_elim _ _ h_A h_negA
  exact ex_falso _ (ArithForm.neg B) h_bot
 have h_neg_imp : SPA_Deduction_Internal s (insert (ArithForm.neg A) Γ) (ArithForm.neg (ArithForm.impl A (ArithForm.neg B))) :=
  SPA_Deduction_Internal.weaken Γ _ _ h (Set.subset_insert _ _)
 exact SPA_Deduction_Internal.neg_elim _ _ h_imp h_neg_imp

lemma and_elim_left_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) (A B : ArithForm s)
  (h : and_form A B ∈ Δ) : A ∈ Δ :=
 maximal_consistent_closed_under_deduction Δ h_max h_con _ (and_elim_left Δ A B (SPA_Deduction_Internal.hyp Δ _ h))

lemma and_elim_right_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) (A B : ArithForm s)
  (h : and_form A B ∈ Δ) : B ∈ Δ :=
 maximal_consistent_closed_under_deduction Δ h_max h_con _ (and_elim_right Δ A B (SPA_Deduction_Internal.hyp Δ _ h))

lemma or_resolve_left {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) (A B : ArithForm s)
  (h_or : or_form A B ∈ Δ) (h_negB : ArithForm.neg B ∈ Δ) : A ∈ Δ := by
 cases h_max A with
 | inl hA => exact hA
 | inr h_negA =>
  have hB : B ∈ Δ := maximal_consistent_imp_closed Δ h_max h_con (ArithForm.neg A) B h_negA h_or
  have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
   SPA_Deduction_Internal.neg_elim Δ B (SPA_Deduction_Internal.hyp Δ B hB) (SPA_Deduction_Internal.hyp Δ (ArithForm.neg B) h_negB)
```

```
  exact False.elim (h_con h_bot)


lemma neg_and_intro_left {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) (A B : ArithForm s)
  (h : ArithForm.neg A ∈ Δ) : ArithForm.neg (and_form A B) ∈ Δ := by
 have h_imp_in : ArithForm.impl A (ArithForm.neg B) ∈ Δ := impl_of_neg_Δ h_max h_con A (ArithForm.neg B) h
 apply maximal_consistent_closed_under_deduction Δ h_max h_con
 apply SPA_Deduction_Internal.neg_intro
 have h_imp : SPA_Deduction_Internal s (insert (and_form A B) Δ) (ArithForm.impl A (ArithForm.neg B)) :=
  SPA_Deduction_Internal.weaken Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_imp_in) (Set.subset_insert _ _)
 have h_neg_imp : SPA_Deduction_Internal s (insert (and_form A B) Δ) (ArithForm.neg (ArithForm.impl A (ArithForm.neg B))) :=
  SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)
 exact SPA_Deduction_Internal.neg_elim _ _ h_imp h_neg_imp


lemma neg_and_intro_right {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) (A B : ArithForm s)
  (h : ArithForm.neg B ∈ Δ) : ArithForm.neg (and_form A B) ∈ Δ := by
 have h_imp_in : ArithForm.impl A (ArithForm.neg B) ∈ Δ := by
  apply maximal_consistent_closed_under_deduction Δ h_max h_con
  apply SPA_Deduction_Internal.imp_intro
  apply SPA_Deduction_Internal.weaken Δ
  · exact SPA_Deduction_Internal.hyp Δ _ h
  · exact Set.subset_insert _ _
 apply maximal_consistent_closed_under_deduction Δ h_max h_con
 apply SPA_Deduction_Internal.neg_intro
 have h_imp : SPA_Deduction_Internal s (insert (and_form A B) Δ) (ArithForm.impl A (ArithForm.neg B)) :=
```

SPA_Deduction_Internal.weaken Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_imp_in) (Set.subset_insert _ _)

have h_neg_imp : SPA_Deduction_Internal s (insert (and_form A B) Δ) (ArithForm.neg (ArithForm.impl A (ArithForm.neg B))) :=

SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)

exact SPA_Deduction_Internal.neg_elim _ _ h_imp h_neg_imp

lemma neg_neg_intro_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con : Consistent Δ) (A : ArithForm s)

(h : A ∈ Δ) : ArithForm.neg (ArithForm.neg A) ∈ Δ := by

apply maximal_consistent_closed_under_deduction Δ h_max h_con

apply SPA_Deduction_Internal.neg_intro

have h1 : SPA_Deduction_Internal s (insert (ArithForm.neg A) Δ) A :=

SPA_Deduction_Internal.weaken Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h) (Set.subset_insert _ _)

have h2 : SPA_Deduction_Internal s (insert (ArithForm.neg A) Δ) (ArithForm.neg A) :=

SPA_Deduction_Internal.hyp _ _ (Or.inl rfl)

exact SPA_Deduction_Internal.neg_elim _ _ h1 h2

Lemas sobre la fórmula de orden y funciones de emparejamiento

Se establecen cuatro lemas que vinculan la sintaxis de la fórmula de orden (`lt_form`) con propiedades demostrables en Δ, y que expresan internamente las propiedades algebraicas de la función de emparejamiento de Gödel.

Lema `NotOccurForm_lt_form_closed`: Variables ligadas en la fórmula de orden

Establece que la variable v (distinta de 3) no ocurre libre en la fórmula de orden estricto entre dos numerales con variable testigo 3:

$$ \forall a, b \in \mathbb{N},\ \forall v \neq 3,\ \text{NotFreeForm } } v\ (\overline{a} < \overline{b})_3 $$

donde el subíndice 3 indica que la variable ligada existencial es la número 3. La demostración verifica que todos los componentes de la fórmula (numerales, suma, sucesor) son cerrados, y que la única variable que podría aparecer libre es la 3, que está ligada por el cuantificador existencial.

Lema `lt_form_3_to_1`: Cambio de variable testigo en la fórmula de orden

Establece que si la fórmula de orden con variable testigo 3 es satisfecha por algún testigo cerrado, entonces también lo es con variable testigo 1, instanciada con el término x:

$$ \Delta \vdash \exists v_3\ (\overline{M} = x + S(v_3)) \implies \Delta \vdash \exists v_1\ (\overline{M} = x + S(v_1)) $$

La demostración utiliza la propiedad de Henkin para extraer el testigo w de la primera fórmula existencial, y luego reintroduce el existencial con variable 1 usando el mismo testigo w. La clave técnica es la simplificación de las sustituciones:

tanto la sustitución sobre v_3 como sobre v_1 producen la misma igualdad $\overline{M} = x + S(w)$, ya que los numerales y los términos cerrados son inmunes a las sustituciones.

Lema `lt_form_intro_of_lt`: Introducción de la fórmula de orden a partir de una desigualdad externa

Establece que si $a < b$ en el metalenguaje, entonces la fórmula correspondiente en el lenguaje objeto es demostrable en Δ:

$$ \forall a, b \in \mathbb{N},\ a < b \implies \Delta \vdash \overline{a} < \overline{b} $$

La demostración construye explícitamente un testigo para la diferencia. Dado que $a < b$, existe $k = b - a - 1$ tal que $b = a + (k + 1)$. Utilizando el homomorfismo de la adición (`natToTerm_add_hom`), se obtiene $\overline{b} = \overline{a} + \overline{k + 1} = \overline{a} + S(\overline{k}) \in \Delta$. Esta igualdad, tras la simplificación de la sustitución, proporciona la instancia requerida para introducir el cuantificador existencial y obtener $\exists v_3\ (\overline{b} = \overline{a} + S(v_3))$, que es exactamente $\overline{a} < \overline{b}$.

Lemas `eq_pair_case_lt` y `eq_pair_case_ge`: Propiedades algebraicas de la función de emparejamiento

Expresan internamente en Δ las igualdades que definen la función de emparejamiento de Gödel según los dos casos de la definición:

Caso $a < b$ (primer caso de `PairForm`):

$$ \Delta \vdash \overline{b^2 + a} = \overline{b} \cdot \overline{b} + \overline{a} $$

Caso $\neg(a < b)$ (segundo caso de `PairForm`):

$$ \Delta \vdash \overline{a^2 + a + b} = (\overline{a} \cdot \overline{a} + \overline{a}) + \overline{b} $$

Ambos lemas se demuestran combinando los homomorfismos de la adición y la multiplicación con las reglas de congruencia. La estrategia consiste en descomponer la expresión del lado izquierdo utilizando `natToTerm_add_hom` y `natToTerm_mul_hom`, y luego reconstruir el lado derecho mediante aplicaciones sucesivas de `eq_add_Δ` y `eq_trans_Δ`.

Significado para la representabilidad del emparejamiento

Estos últimos dos lemas son pasos preparatorios para demostrar que la función de emparejamiento de Gödel (definida por `PairForm` en el lenguaje objeto) es representable y se comporta correctamente respecto a los numerales. Las igualdades establecidas son exactamente las que aparecen en los dos casos de la definición de `PairForm`:

$$ \text{PairForm}(a, b, c) \equiv (a < b \land c = b^2 + a) \lor (\neg(a < b) \land c = a^2 + a + b) $$

Los lemas `eq_pair_case_lt` y `eq_pair_case_ge` proporcionan las igualdades internas necesarias para verificar que, dados a y b, el código de par calculado externamente como $\text{pair}(a, b)$ satisface la fórmula `PairForm` en Δ. Esto es un ingrediente esencial para la representabilidad de la codificación de secuencias y, en última instancia, para la aritmetización completa de la sintaxis en el sistema SPA.

```
lemma NotOccurForm_lt_form_closed {s : Nat} (a b : Nat) (v : Nat) (hv : v ≠ 3) :
  NotOccurForm v (lt_form (s := s) (@natToTerm s a) (@natToTerm s b) 3) := by
 dsimp [lt_form, exists_form, ArithForm.exists_form, NotOccurForm]
 refine ⟨hv, ?_⟩
 refine ⟨NotOccur_of_isClosed (@natToTerm s b) (isClosed_natToTerm b) v, ?_⟩
 apply NotOccur.add
 · exact NotOccur_of_isClosed (@natToTerm s a) (isClosed_natToTerm a) v
 · apply NotOccur.succ
  exact NotOccur.var 3 hv


lemma lt_form_3_to_1 {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)
  (x : TermClosed s) (M : Nat)
  (h : exists_form 3 (ArithForm.eq (@natToTerm s M) (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var 3)))) ∈ Δ) :
  substForm 0 x.val (lt_form (ArithTerm.var 0) (@natToTerm s M) 1) ∈ Δ := by
 obtain ⟨w, hw⟩ := h_henkin 3 (ArithForm.eq (@natToTerm s M) (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var 3)))) h
 have hw_simp : substForm 3 w.val (ArithForm.eq (@natToTerm s M) (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var 3)))) =
       ArithForm.eq (@natToTerm s M) (ArithTerm.add x.val (ArithTerm.succ w.val)) := by
  dsimp [substForm, substTerm]
  have h_M := substTerm_fresh (@natToTerm s M) 3 w.val (NotOccur_of_isClosed (@natToTerm s M) (isClosed_natToTerm M) 3)
  have h_x := substTerm_fresh x.val 3 w.val (NotOccur_of_isClosed x.val x.property 3)
  rw [h_M, h_x]
```

```
  rw [hw_simp] at hw

  rw [substForm_lt_form x (@natToTerm s M) (isClosed_natToTerm M)]

  have h_subst : substForm 1 w.val (ArithForm.eq (@natToTerm s M) (ArithTerm.add
x.val (ArithTerm.succ (ArithTerm.var 1)))) =

          ArithForm.eq (@natToTerm s M) (ArithTerm.add x.val (ArithTerm.succ
w.val)) := by

    dsimp [substForm, substTerm]

    have h_M := substTerm_fresh (@natToTerm s M) 1 w.val (NotOccur_of_isClosed
(@natToTerm s M) (isClosed_natToTerm M) 1)

    have h_x := substTerm_fresh x.val 1 w.val (NotOccur_of_isClosed x.val x.property 1)

    rw [h_M, h_x]

  have hw_rew : substForm 1 w.val (ArithForm.eq (@natToTerm s M) (ArithTerm.add
x.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ := by

    rwa [h_subst]

  have h_free : IsFreeFor 1 w.val (ArithForm.eq (@natToTerm s M) (ArithTerm.add
x.val (ArithTerm.succ (ArithTerm.var 1)))) :=

    IsFreeFor_closed _ 1 w

  have h_exists := exists_intro Δ 1 (ArithForm.eq (@natToTerm s M) (ArithTerm.add
x.val (ArithTerm.succ (ArithTerm.var 1)))) w.val h_free (SPA_Deduction_Internal.hyp
Δ _ hw_rew)

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists


lemma lt_form_intro_of_lt {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ)

    (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

    (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

    (a b : Nat) (h_lt : a < b) :

    lt_form (s := s) (@natToTerm s a) (@natToTerm s b) 3 ∈ Δ := by

  let k := b - a - 1

  have h_eq_b : b = a + (k + 1) := by omega

  have h_hom := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ a (k + 1)
```

```
  have h_b_eq : @natToTerm s b = @natToTerm s (a + (k + 1)) := by rw [h_eq_b]

  rw [← h_b_eq] at h_hom

  have h_subst : substForm 3 (@natToTerm s k) (ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var 3)))) =

        ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (@natToTerm s k))) := by

    dsimp [substForm, substTerm]

    have h_b' := substTerm_fresh (@natToTerm s b) 3 (@natToTerm s k) (NotOccur_of_isClosed (@natToTerm s b) (isClosed_natToTerm b) 3)

    have h_a' := substTerm_fresh (@natToTerm s a) 3 (@natToTerm s k) (NotOccur_of_isClosed (@natToTerm s a) (isClosed_natToTerm a) 3)

    rw [h_b', h_a']

  have h_hom_rew : substForm 3 (@natToTerm s k) (ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var 3)))) ∈ Δ := by

    rwa [h_subst]

  have h_free : IsFreeFor 3 (@natToTerm s k) (ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var 3)))) :=

    IsFreeFor_closed _ 3 ⟨@natToTerm s k, isClosed_natToTerm k⟩

  have h_exists := exists_intro Δ 3 (ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var 3)))) (@natToTerm s k) h_free (SPA_Deduction_Internal.hyp Δ _ h_hom_rew)

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists


lemma eq_pair_case_lt {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

    (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

    (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

    (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

    (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

    (a b : Nat) :

    ArithForm.eq (@natToTerm s (b * b + a)) (ArithTerm.add (ArithTerm.mul (@natToTerm s b) (@natToTerm s b)) (@natToTerm s a)) ∈ Δ := by
```

have h_add := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ (b * b) a

have h_mul := natToTerm_mul_hom Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ b b

have h_refl := eq_refl_in_Δ Δ h_max h_con (@natToTerm s a)

have h_congr := eq_add_Δ h_max h_con h_mul h_refl

exact eq_trans_Δ h_max h_con h_add h_congr

lemma eq_pair_case_ge {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

(h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

(a b : Nat) :

ArithForm.eq (@natToTerm s (a * a + a + b)) (ArithTerm.add (ArithTerm.add (ArithTerm.mul (@natToTerm s a) (@natToTerm s a)) (@natToTerm s a)) (@natToTerm s b)) ∈ Δ := by

have h_add1 := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ (a * a + a) b

have h_add2 := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ (a * a) a

have h_mul := natToTerm_mul_hom Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ a a

have h_refl_a := eq_refl_in_Δ Δ h_max h_con (@natToTerm s a)

have h_congr_mul := eq_add_Δ h_max h_con h_mul h_refl_a

have h_trans1 := eq_trans_Δ h_max h_con h_add2 h_congr_mul

have h_refl_b := eq_refl_in_Δ Δ h_max h_con (@natToTerm s b)

have h_congr_add := eq_add_Δ h_max h_con h_trans1 h_refl_b

exact eq_trans_Δ h_max h_con h_add1 h_congr_add

Lemas de contradicción y negación para la fórmula de orden, y sustitución en `PairForm`

Se demuestran tres lemas que completan la infraestructura para razonar sobre la fórmula de orden y la fórmula de emparejamiento dentro de Δ.

Lema `lt_form_contradiction`: Contradicción entre orden interno y orden externo

Establece que no puede ocurrir simultáneamente que $a \geq b$ en el metalenguaje y que Δ demuestre $\overline{a} < \overline{b}$:

$$ \forall a, b \in \mathbb{N},\ a \geq b \land \Delta \vdash \overline{a} < \overline{b} \implies \bot $$

Estructura de la demostración

1. **Conversión de variable testigo:** Se aplica `lt_form_3_to_1` para transformar la fórmula con variable testigo 3 en la fórmula equivalente con variable testigo 1, manteniendo el mismo término $x = \overline{a}$.

2. **Aplicación del teorema de acotación:** Se invoca `lt_numeral_bounded`, que establece que si $\overline{a} < \overline{b}$ en Δ, entonces existe $k < b$ tal que $\Delta \vdash \overline{a} = \overline{k}$.

3. **Contradicción aritmética:** Dado que $a \geq b$ y $k < b$, se tiene $a \neq k$. Por `prove_distinct_numerals_Δ`, $\Delta \vdash \neg(\overline{a} = \overline{k})$.

Pero el paso anterior proporciona $\Delta \vdash \overline{a} = \overline{k}$, lo cual contradice la consistencia de Δ.

Intuición y necesidad

Este lema establece la corrección de la fórmula de orden interno respecto a la interpretación pretendida: si externamente a es mayor o igual que b, es imposible que internamente se demuestre $a < b$. Es la contraparte negativa de `lt_form_intro_of_lt`, y juntos garantizan que la noción interna de orden coincide exactamente con la externa para los numerales.

Lema `neg_lt_form_of_ge`: Negación interna del orden a partir de una desigualdad externa

Establece que si $a \geq b$ externamente, entonces Δ demuestra la negación de $\overline{a} < \overline{b}$:

$$ \forall a, b \in \mathbb{N},\ a \geq b \implies \Delta \vdash \neg(\overline{a} < \overline{b}) $$

La demostración es directa por maximalidad de Δ: por el lema anterior, $\overline{a} < \overline{b}$ no puede pertenecer a Δ (pues generaría una contradicción), y por maximalidad su negación debe pertenecer. Este lema es la herramienta principal para decidir internamente desigualdades entre numerales.

Lema `subst_PairForm`: Simplificación de la sustitución en la fórmula de emparejamiento

Establece la forma explícita que adopta la fórmula `PairForm` cuando sus tres variables libres se instancian con numerales:

$$ \text{PairForm}[v_0 := \overline{a},\ v_1 := \overline{b},\ v_2 := P] = $$

$$ (\overline{a} < \overline{b} \land P = \overline{b}^2 + \overline{a}) \lor (\neg(\overline{a} < \overline{b}) \land P = \overline{a}^2 + \overline{a} + \overline{b}) $$

Estructura de la demostración

La prueba consiste en desplegar completamente las definiciones de `PairForm`, `or_form`, `and_form`, `lt_form`, `exists_form`, `substForm` y `substTerm`. La simplificación se basa en tres aplicaciones de `substTerm_fresh`:

- La sustitución de v_0 por $\overline{a}$ es inocua porque las variables 1 y 2 no ocurren en $\overline{a}$ (es un numeral cerrado).

- La sustitución de v_1 por $\overline{b}$ es igualmente inocua.

- La sustitución de v_2 por P preserva las ocurrencias de $\overline{a}$ y $\overline{b}$ en la fórmula porque estas son términos cerrados.

Tras estas simplificaciones, la fórmula colapsa exactamente en la disyunción de los dos casos que definen el emparejamiento de Gödel.

Significado para la representabilidad del emparejamiento

Este lema es fundamental porque proporciona la conexión directa entre la definición sintáctica de `PairForm` en el lenguaje objeto y la función de emparejamiento definida en el metalenguaje. Cuando P es exactamente $\overline{\text{pair}(a, b)}$, la fórmula instanciada se convierte en una afirmación sobre cuál de los dos casos de la definición se aplica, dependiendo de si $a < b$ o no. Junto con los lemas `eq_pair_case_lt` y `eq_pair_case_ge` (que establecen las igualdades algebraicas necesarias) y los lemas de orden (`lt_form_intro_of_lt` y `neg_lt_form_of_ge`, que deciden cuál caso aplica), se tiene toda la maquinaria necesaria para demostrar que `PairForm` representa correctamente la función de emparejamiento en Δ.

```
lemma lt_form_contradiction {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

  (a b : Nat) (h_ge : a ≥ b)

  (h_lt : lt_form (s := s) (@natToTerm s a) (@natToTerm s b) 3 ∈ Δ) : False := by

 have h_lt_1 := lt_form_3_to_1 Δ h_max h_con h_henkin ⟨@natToTerm s a, isClosed_natToTerm a⟩ b h_lt

 obtain ⟨k, hk_lt, hk_eq⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ b ⟨@natToTerm s a, isClosed_natToTerm a⟩ h_lt_1

 have h_ne : a ≠ k := by omega

 have h_neg := prove_distinct_numerals_Δ h_max h_con a k h_ne

 have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=

  SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ hk_eq) (SPA_Deduction_Internal.hyp Δ _ h_neg)

 exact h_con h_bot


lemma neg_lt_form_of_ge {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

  (a b : Nat) (h_ge : a ≥ b) :

  ArithForm.neg (lt_form (s := s) (@natToTerm s a) (@natToTerm s b) 3) ∈ Δ := by

 cases h_max (lt_form (@natToTerm s a) (@natToTerm s b) 3) with

 | inl h_lt =>

  exact False.elim (lt_form_contradiction Δ h_max h_con h_henkin h_add_zero h_add_succ a b h_ge h_lt)
```

```
  | inr h_neg =>
    exact h_neg


lemma subst_PairForm {s : Nat} (a b : Nat) (P : ArithTerm s) :
    substForm 2 P (substForm 1 (@natToTerm s b) (substForm 0 (@natToTerm s a) (PairForm (s := s)))) =
    or_form
      (and_form (lt_form (@natToTerm s a) (@natToTerm s b) 3)
          (ArithForm.eq P (ArithTerm.add (ArithTerm.mul (@natToTerm s b) (@natToTerm s b)) (@natToTerm s a))))
      (and_form (ArithForm.neg (lt_form (@natToTerm s a) (@natToTerm s b) 3))
          (ArithForm.eq P (ArithTerm.add (ArithTerm.add (ArithTerm.mul (@natToTerm s a) (@natToTerm s a)) (@natToTerm s a)) (@natToTerm s b)))) := by
  unfold PairForm
  dsimp [or_form, and_form, lt_form, exists_form, ArithForm.exists_form, substForm, substTerm]
  have h_a0 : substTerm 1 (@natToTerm s b) (@natToTerm s a) = @natToTerm s a :=
    substTerm_fresh (@natToTerm s a) 1 (@natToTerm s b) (NotOccur_of_isClosed (@natToTerm s a) (isClosed_natToTerm a) 1)
  have h_a1 : substTerm 2 P (@natToTerm s a) = @natToTerm s a :=
    substTerm_fresh (@natToTerm s a) 2 P (NotOccur_of_isClosed (@natToTerm s a) (isClosed_natToTerm a) 2)
  have h_b0 : substTerm 2 P (@natToTerm s b) = @natToTerm s b :=
    substTerm_fresh (@natToTerm s b) 2 P (NotOccur_of_isClosed (@natToTerm s b) (isClosed_natToTerm b) 2)
  rw [h_a0, h_a1, h_b0]
```

Teorema principal `pair_representable`: Representabilidad de la función de emparejamiento

Se demuestra el teorema central que establece que la función de emparejamiento de Gödel es representable en cualquier teoría Δ maximal consistente con propiedad de Henkin y que contenga los axiomas de la aritmética. Formalmente, la relación ternaria definida por `PairForm` representa correctamente la función $\text{pair} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$:

$$ \forall a, b \in \mathbb{N}, $$

1. **Condición de gráfico:** $\Delta \vdash \text{PairForm}(\overline{a}, \overline{b}, \overline{\text{pair}(a, b)})$

2. **Condición de funcionalidad:** $\forall P \in \text{TermClosed } s,\ \Delta \vdash \text{PairForm}(\overline{a}, \overline{b}, P) \implies \Delta \vdash P = \overline{\text{pair}(a, b)}$

Estructura de la demostración

La prueba se divide en dos partes, correspondientes a las dos condiciones de representabilidad:

Parte 1: Condición de gráfico

Se debe demostrar que la instancia de `PairForm` con los numerales $\overline{a}$, $\overline{b}$ y $\overline{\text{pair}(a, b)}$ pertenece a Δ. La estrategia distingue dos casos según si $a < b$ o no, reflejando la definición de la función de emparejamiento:

Caso $a < b$ (primer caso de la definición):

1. Se tiene $\text{pair}(a, b) = b^2 + a$ (por definición de `Nat.pair` cuando el primer argumento es menor que el segundo).

2. Por `lt_form_intro_of_lt`, se obtiene $\overline{a} < \overline{b} \in \Delta$.

3. Por `eq_pair_case_lt`, se obtiene $\overline{b^2 + a} = \overline{b}^2 + \overline{a} \in \Delta$.

4. Mediante `and_intro` se combinan ambas para obtener la primera rama de la disyunción: $(\overline{a} < \overline{b}) \land (P = \overline{b}^2 + \overline{a}) \in \Delta$, donde $P = \overline{b^2 + a}$.

5. Por `or_intro_left_Δ`, esta conjunción implica la disyunción completa.

6. Se reescribe $\overline{\text{pair}(a, b)}$ como $\overline{b^2 + a}$ utilizando la igualdad del paso 1, y se aplica `subst_PairForm` para obtener exactamente la fórmula requerida.

Caso $a \geq b$ (segundo caso de la definición):

1. Se tiene $\text{pair}(a, b) = a^2 + a + b$ (por definición de `Nat.pair` cuando $a \geq b$).

2. Por `neg_lt_form_of_ge`, se obtiene $\neg(\overline{a} < \overline{b}) \in \Delta$.

3. Por `eq_pair_case_ge`, se obtiene $\overline{a^2 + a + b} = \overline{a}^2 + \overline{a} + \overline{b} \in \Delta$.

4. Mediante `and_intro` se combinan para obtener la segunda rama: $\neg(\overline{a} < \overline{b}) \land (P = \overline{a}^2 + \overline{a} + \overline{b}) \in \Delta$.

5. Por `or_intro_right_Δ`, esta conjunción implica la disyunción completa.

6. Se reescribe análogamente al caso anterior.

Parte 2: Condición de funcionalidad

Se asume que para algún término cerrado P, se tiene $\text{PairForm}(\overline{a}, \overline{b}, P) \in \Delta$, y se debe demostrar $P = \overline{\text{pair}(a, b)} \in \Delta$. Nuevamente se distinguen dos casos:

Caso $a < b$:

1. Se sabe que $\text{pair}(a, b) = b^2 + a$, y se reescribe `subst_PairForm` en la hipótesis.

2. Por `lt_form_intro_of_lt`, se tiene $\overline{a} < \overline{b} \in \Delta$. Por doble negación (`neg_neg_intro_Δ`), $\neg\neg(\overline{a} < \overline{b}) \in \Delta$.

3. Por `neg_and_intro_left`, $\neg(\neg(\overline{a} < \overline{b}) \land (P = \overline{a}^2 + \overline{a} + \overline{b})) \in \Delta$. Esto niega la segunda rama de la disyunción.

4. Por `or_resolve_left` (silogismo disyuntivo), se deduce que la primera rama debe ser cierta: $(\overline{a} < \overline{b}) \land (P = \overline{b}^2 + \overline{a}) \in \Delta$.

5. Por `and_elim_right_Δ`, se extrae $P = \overline{b}^2 + \overline{a} \in \Delta$.

6. Por `eq_pair_case_lt` y simetría, $\overline{b}^2 + \overline{a} = \overline{b^2 + a} \in \Delta$.

7. Por transitividad, $P = \overline{b^2 + a} = \overline{\text{pair}(a, b)} \in \Delta$.

Caso $a \geq b$:

1. Se sabe que $\text{pair}(a, b) = a^2 + a + b$.

2. Por `neg_lt_form_of_ge`, se tiene $\neg(\overline{a} < \overline{b}) \in \Delta$.

3. Por `neg_and_intro_left`, se niega la primera rama de la disyunción.

4. La hipótesis `h_in` es una disyunción $X \lor Y$ con $\neg X \in \Delta$. Dado que $X \lor Y$ se define como $\neg X \to Y$, por modus ponens (`maximal_consistent_imp_closed`) se obtiene $Y \in \Delta$.

5. Por `and_elim_right_Δ`, se extrae $P = \overline{a}^2 + \overline{a} + \overline{b} \in \Delta$.

6. Por `eq_pair_case_ge` y simetría, $\overline{a}^2 + \overline{a} + \overline{b} = \overline{a^2 + a + b} \in \Delta$.

7. Por transitividad, $P = \overline{a^2 + a + b} = \overline{\text{pair}(a, b)} \in \Delta$.

Significado del teorema

`pair_representable` es un resultado fundamental en la aritmetización de la sintaxis. Establece que la función de emparejamiento de Gödel —el bloque básico para codificar secuencias finitas— es representable en el sistema SPA. La fórmula `PairForm` captura exactamente la definición por casos de la función de Cantor $\text{pair}(a, b)$:

$$ \text{pair}(a, b) = \begin{cases} b^2 + a & \text{si } a < b \\ a^2 + a + b & \text{si } a \geq b \end{cases} $$

La demostración pone de manifiesto la potencia de la infraestructura desarrollada: los lemas de orden (`lt_form_intro_of_lt`, `neg_lt_form_of_ge`) permiten decidir internamente cuál de los dos casos aplica; los lemas algebraicos (`eq_pair_case_lt`, `eq_pair_case_ge`) proporcionan las igualdades numéricas necesarias; y los lemas lógicos (`or_resolve_left`, `and_elim_right_Δ`, etc.) permiten manipular la estructura disyuntiva de la fórmula. La combinación de todos estos elementos produce una demostración elegante y modular de la representabilidad del emparejamiento, que servirá como base para la representabilidad de secuencias finitas y, en última instancia, para la codificación completa de la sintaxis del sistema SPA.

```
--
============================================================================

-- DEMOSTRACIÓN TEOREMA PRINCIPAL: pair_representable

--
============================================================================


lemma pair_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con
: Consistent Δ) (h_henkin : HenkinSet Δ)

   (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

   (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

   (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

   (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

   (a b : Nat) :

   (substForm 2 (@natToTerm s (pair a b)) (substForm 1 (@natToTerm s b)
(substForm 0 (@natToTerm s a) PairForm)) ∈ Δ) ∧

   (∀ (P : TermClosed s),

    substForm 2 P.val (substForm 1 (@natToTerm s b) (substForm 0 (@natToTerm s
a) PairForm)) ∈ Δ →

    ArithForm.eq P.val (@natToTerm s (pair a b)) ∈ Δ) := by

 constructor

 · -- Gráfico

   by_cases h_lt : a < b

   · have h1 := lt_form_intro_of_lt Δ h_max h_con h_add_zero h_add_succ a b h_lt

    have h2 := eq_pair_case_lt Δ h_max h_con h_add_zero h_add_succ h_mul_zero
h_mul_succ a b

    have h_and := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h1)
(SPA_Deduction_Internal.hyp Δ _ h2)
```

```
    have h_and_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and

    have h_or := or_intro_left_Δ h_max h_con _ (and_form (ArithForm.neg (lt_form (@natToTerm s a) (@natToTerm s b) 3)) (ArithForm.eq (@natToTerm s (pair a b)) (ArithTerm.add (ArithTerm.add (ArithTerm.mul (@natToTerm s a) (@natToTerm s a)) (@natToTerm s a)) (@natToTerm s b)))) h_and_in

    have h_pair_b : pair a b = b * b + a := by

      simp [pair, Nat.pair, h_lt]

    rw [h_pair_b] at h_or ⊢

    rw [subst_PairForm a b (@natToTerm s (b * b + a))]

    exact h_or

  · have h_ge : a ≥ b := by omega

    have h1 := neg_lt_form_of_ge Δ h_max h_con h_henkin h_add_zero h_add_succ a b h_ge

    have h2 := eq_pair_case_ge Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ a b

    have h_and := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h1) (SPA_Deduction_Internal.hyp Δ _ h2)

    have h_and_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and

    have h_or := or_intro_right_Δ h_max h_con (and_form (lt_form (@natToTerm s a) (@natToTerm s b) 3) (ArithForm.eq (@natToTerm s (pair a b)) (ArithTerm.add (ArithTerm.mul (@natToTerm s b) (@natToTerm s b)) (@natToTerm s a)))) _ h_and_in

    have h_pair_b : pair a b = a * a + a + b := by

      simp [pair, Nat.pair, h_lt]

    rw [h_pair_b] at h_or ⊢

    rw [subst_PairForm a b (@natToTerm s (a * a + a + b))]

    exact h_or

  · -- Funcionalidad

   intro P h_in

   by_cases h_lt : a < b
```

```
  · have h_pair_b : pair a b = b * b + a := by
      simp [pair, Nat.pair, h_lt]
    rw [subst_PairForm a b P.val] at h_in
    have h_lt_in := lt_form_intro_of_lt Δ h_max h_con h_add_zero h_add_succ a b h_lt
    have h_neg_neg_lt := neg_neg_intro_Δ h_max h_con _ h_lt_in
    have h_neg_Y := neg_and_intro_left h_max h_con _ (ArithForm.eq P.val (ArithTerm.add (ArithTerm.add (ArithTerm.mul (@natToTerm s a) (@natToTerm s a)) (@natToTerm s a)) (@natToTerm s b))) h_neg_neg_lt
    have h_X := or_resolve_left h_max h_con _ _ h_in h_neg_Y
    have h_eq1 := and_elim_right_Δ h_max h_con _ _ h_X
    have h_eq2 := eq_pair_case_lt Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ a b
    have h_eq2_symm := eq_symm_Δ h_max h_con h_eq2
    have h_final := eq_trans_Δ h_max h_con h_eq1 h_eq2_symm
    rw [h_pair_b]
    exact h_final
  · have h_ge : a ≥ b := by omega
    have h_pair_b : pair a b = a * a + a + b := by
      simp [pair, Nat.pair, h_lt]
    rw [subst_PairForm a b P.val] at h_in
    have h_neg_lt := neg_lt_form_of_ge Δ h_max h_con h_henkin h_add_zero h_add_succ a b h_ge
    have h_neg_X := neg_and_intro_left h_max h_con _ (ArithForm.eq P.val (ArithTerm.add (ArithTerm.mul (@natToTerm s b) (@natToTerm s b)) (@natToTerm s a))) h_neg_lt
    have h_Y := maximal_consistent_imp_closed Δ h_max h_con _ _ h_neg_X h_in
    have h_eq1 := and_elim_right_Δ h_max h_con _ _ h_Y
    have h_eq2 := eq_pair_case_ge Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ a b
    have h_eq2_symm := eq_symm_Δ h_max h_con h_eq2
    have h_final := eq_trans_Δ h_max h_con h_eq1 h_eq2_symm
```

rw [h_pair_b]

exact h_final

Fórmula de composición generalizada `compose_formula`

Se define la fórmula que representa la composición de dos relaciones funcionales representables. Dadas dos fórmulas φ_f y φ_g (cada una con dos variables libres v_0 y v_1), la composición $\varphi_{f \circ g}$ se define como:

$$ \text{compose_formula}(\varphi_f, \varphi_g, w) := \exists v_w\ (\varphi_g(v_0, v_w) \land \varphi_f(v_w, v_1)) $$

donde w es una variable fresca que actuará como variable ligada existencial para el resultado intermedio de la composición.

Intuición y necesidad

Esta fórmula captura la noción matemática de composición de funciones: $y = f(g(x))$ si y solo si existe un valor intermedio z tal que $z = g(x)$ e $y = f(z)$. En el contexto de la representabilidad, si φ_f representa una función f y φ_g representa una función g, entonces `compose_formula` representa la función compuesta $f \circ g$. La variable w debe elegirse de manera que no aparezca libre en φ_f ni en φ_g para evitar capturas de variables. Esta construcción es esencial para la aritmetización, ya que permite construir fórmulas representadoras para funciones complejas a partir de fórmulas más simples.

Lemas de conmutación de sustituciones generales

Se establecen cuatro lemas técnicos que generalizan las propiedades de conmutación de sustituciones a contextos con variables ligadas y composiciones de sustituciones.

Lema `substTerm_var_fresh`: Sustitución sobre variable distinta

$$ \forall v, w \in \mathbb{N},\ v \neq w \implies \text{substTerm}(v, t, \text{var } w) = \text{var } w $$

La sustitución de v por t en una variable w distinta de v no tiene efecto. La demostración es inmediata por la definición de `substTerm`, que utiliza un condicional sobre si la variable objetivo coincide con la variable de sustitución.

Lema `substForm_forall_form_fresh`: Sustitución que no afecta a la variable ligada

$$ \forall n \neq v,\ \text{substForm}(v, t, \forall v_n\ \varphi) = \forall v_n\ (\text{substForm}(v, t, \varphi)) $$

Cuando la variable de sustitución v es distinta de la variable ligada n, la sustitución simplemente atraviesa el cuantificador y se aplica al cuerpo de la fórmula. La demostración despliega la definición de `substForm` y utiliza que la condición $v = n$ es falsa, por lo que se toma la rama recursiva.

Lema `substForm_exists_form_fresh`: Sustitución que no afecta a la variable ligada existencial

$$ \forall n \neq v,\ \text{substForm}(v, t, \exists v_n\ \varphi) = \exists v_n\ (\text{substForm}(v, t, \varphi)) $$

Análogo al lema anterior para el cuantificador existencial (definido como $\neg \forall \neg$). La demostración expande la definición de `exists_form` en términos de `forall_form` y `neg`, aplica el lema anterior, y simplifica.

Lema `substTerm_substTerm_term_comm`: Conmutación de sustituciones con variable intermedia fresca

$$ \forall t, u \in \text{ArithTerm } s,\ \forall v, w \in \mathbb{N},\ \text{NotOccur } w\ t \implies \text{substTerm}(w, u, \text{substTerm}(v, \text{var } w, t)) = \text{substTerm}(v, u, t) $$

Este lema establece una propiedad fundamental de conmutación: sustituir v por una variable fresca w y luego sustituir w por u es equivalente a sustituir directamente v por u, siempre que w no ocurra originalmente en t.

Estructura de la demostración

La prueba procede por inducción estructural sobre t, cubriendo todos los constructores de `ArithTerm`:

- **Caso `var n`:** Se analiza si $n = v$ o no. Si coincide, la sustitución de v por $\text{var } w$ produce $\text{var } w$, y la sustitución subsiguiente de w por u produce u, que coincide con $\text{substTerm}(v, u, \text{var } v)$. Si $n \neq v$, ambas ramas de la igualdad preservan $\text{var } n$, pues w no puede coincidir con n (por la hipótesis `NotOccur w (var n)`, que fuerza $w \neq n$).

- **Casos base (`const`, `henkin`, `zero`):** Triviales, pues la sustitución no afecta a términos sin variables.

- **Casos recursivos (`succ`, `add`, `mul`, `exp`, `pred`):** Se aplica la hipótesis de inducción a cada subcomponente, utilizando que la condición `NotOccur w t` se descompone en condiciones análogas para los subtérminos (por inversión de los constructores de `NotOccur`).

Significado para la composición de fórmulas

Este último lema es crucial para la demostración de que `compose_formula` efectivamente representa la composición de funciones. Cuando se instancia la fórmula de composición con códigos concretos, aparecen sustituciones encadenadas de la forma $\text{substTerm}(w, u, \text{substTerm}(v, \text{var } w, t))$, y este lema permite colapsarlas en una única sustitución $\text{substTerm}(v, u, t)$, simplificando drásticamente las expresiones resultantes. La condición de frescura $\text{NotOccur } w\ t$ se satisface automáticamente cuando w se elige como una variable nueva que no aparece en t, lo cual es siempre posible gracias al suministro ilimitado de variables en el lenguaje.

--
==========

-- DEFINICIÓN DE LA FÓRMULA DE COMPOSICIÓN GENERALIZADA

--
==========

/--

Fórmula de SPA que representa la composición de dos fórmulas funcionales representables.

φ_fg(x, y) := ∃ z, φ_g(x, z) ∧ φ_f(z, y)

Utiliza la variable ligada interna `w` especificada como parámetro.

-/

def compose_formula {s : Nat} (φ_f φ_g : ArithForm s) (w : Nat) : ArithForm s :=

exists_form w (and_form (substForm 1 (ArithTerm.var w) φ_g) (substForm 0 (ArithTerm.var w) φ_f))

--
==========

-- LEMAS ADICIONALES DE CONMUTACIÓN DE SUSTITUCIONES GENERALES

--
==========

lemma substTerm_var_fresh {s : Nat} (v : Nat) (t : ArithTerm s) (w : Nat) (hw : w ≠ v) :

substTerm v t (ArithTerm.var (s := s) w) = ArithTerm.var (s := s) w := by

dsimp [substTerm]

have h_ne : ¬ (v = w) := by

intro h_eq

exact hw h_eq.symm

rw [if_neg h_ne]

lemma substForm_forall_form_fresh {s : Nat} (v : Nat) (t : ArithTerm s) (n : Nat) (f : ArithForm s) (hv : n ≠ v) :

substForm v t (ArithForm.forall_form n f) = ArithForm.forall_form n (substForm v t f) := by

dsimp [substForm]

have h_ne : ¬ (v = n) := by

intro h_eq

```
  exact hv h_eq.symm
 rw [if_neg h_ne]

lemma substForm_exists_form_fresh {s : Nat} (v : Nat) (t : ArithTerm s) (n : Nat) (f :
ArithForm s) (hv : n ≠ v) :
  substForm v t (exists_form n f) = exists_form n (substForm v t f) := by
 change ArithForm.neg (substForm v t (ArithForm.forall_form n (ArithForm.neg f))) =
      ArithForm.neg (ArithForm.forall_form n (ArithForm.neg (substForm v t f)))
 rw [substForm_forall_form_fresh v t n (ArithForm.neg f) hv]
 rfl

lemma substTerm_substTerm_term_comm {s : Nat} (t : ArithTerm s) (v w : Nat) (u :
ArithTerm s) (hw : NotOccur w t) :
  substTerm w u (substTerm v (ArithTerm.var w) t) = substTerm v u t := by
 induction t with
 | var n =>
  simp only [substTerm_var]
  split_ifs with h1
  · subst h1
   simp only [substTerm_var]
   split_ifs with _h2
   · rfl
   · contradiction
  · simp only [substTerm_var]
   split_ifs with _h2
   · cases hw with | var _ h_ne => contradiction
   · rfl
 | const c' => rfl
 | henkin c' => rfl
```

```
| zero => rfl
| succ t' ih =>
  cases hw with | succ _ h_sub =>
  dsimp [substTerm]
  rw [ih h_sub]
| add t1 t2 ih1 ih2 =>
  cases hw with | add _ _ h1 h2 =>
  dsimp [substTerm]
  rw [ih1 h1, ih2 h2]
| mul t1 t2 ih1 ih2 =>
  cases hw with | mul _ _ h1 h2 =>
  dsimp [substTerm]
  rw [ih1 h1, ih2 h2]
| exp t1 t2 ih1 ih2 =>
  cases hw with | exp _ _ h1 h2 =>
  dsimp [substTerm]
  rw [ih1 h1, ih2 h2]
| pred t' ih =>
  cases hw with | pred _ h_sub =>
  dsimp [substTerm]
  rw [ih h_sub]
```

Lema `substForm_substForm_term_comm`: Conmutación de sustituciones en fórmulas con variable intermedia fresca

Se demuestra el análogo del lema `substTerm_substTerm_term_comm` para el nivel de las fórmulas. Este lema establece que la sustitución secuencial de una variable v por una variable fresca w, seguida de la sustitución de w por un término u, es equivalente a la sustitución directa de v por u, siempre que w no ocurra libre en la fórmula original:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v, w \in \mathbb{N},\ \forall u \in \text{ArithTerm } s,\ \text{NotOccurForm } w\ \varphi \implies $$

$$ \text{substForm}(w, u, \text{substForm}(v, \text{var } w, \varphi)) = \text{substForm}(v, u, \varphi) $$

Estructura de la demostración

La prueba procede por casos sobre la estructura de φ, cubriendo exhaustivamente los ocho constructores de `ArithForm`:

- **Casos atómicos (`bottom`, `eq`, `list_nth_eq`):** Para `bottom` es trivial (no hay variables). Para `eq` y `list_nth_eq`, la condición `NotOccurForm w φ` se descompone en condiciones `NotOccur w` sobre los subtérminos, y la demostración se reduce al lema `substTerm_substTerm_term_comm` aplicado a cada subtérmino.

- **Conectivos proposicionales (`impl`, `neg`):** La condición de no ocurrencia se propaga a las subfórmulas, y las hipótesis de inducción resuelven cada caso.

- **Cuantificador universal (`forall_form n f`):** Este es el caso más delicado. Se distinguen dos subcasos:

  - Si $v = n$ (la variable de sustitución es la variable ligada), entonces la sustitución de v por $\text{var } w$ no afecta a la fórmula (la variable está ligada), y la sustitución posterior de w por u tampoco afecta (porque $w \neq n$ por la hipótesis

`NotOccurForm w (forall_form n f)`). El resultado es la fórmula original, que coincide con `substForm v u` cuando v está ligada.

- Si $v \neq n$, ambas sustituciones atraviesan el cuantificador (pues $w \neq n$ también) y se aplican al cuerpo f, donde la hipótesis de inducción resuelve la igualdad.

- **Operadores estratificados (`provable_op`, `true_op`):** Se reducen al lema para términos, ya que solo contienen un subtérmino.

Significado técnico

Este lema es la contraparte a nivel de fórmulas del lema de conmutación para términos. Es una herramienta indispensable para manipular fórmulas que involucran composición de relaciones, donde típicamente se introduce una variable fresca para representar un valor intermedio y luego se sustituye por un término concreto. La condición `NotOccurForm w φ` garantiza que la variable fresca no interfiere con las variables ligadas ni libres de φ, lo cual se satisface siempre que w se elija suficientemente grande.

Lema `subst_comm_closed`: Conmutación de sustituciones con términos cerrados

Establece que dos sustituciones sobre variables distintas con términos cerrados conmutan entre sí:

$$ \forall \varphi \in \text{ArithForm } s,\ \forall v \neq w,\ \forall t, u \in \text{TermClosed } s, $$

$$ \text{substForm}(v, t.\text{val}, \text{substForm}(w, u.\text{val}, \varphi)) = \text{substForm}(w, u.\text{val}, \text{substForm}(v, t.\text{val}, \varphi)) $$

Estructura de la demostración

La prueba se apoya en el lema general de conmutación de sustituciones `substForm_comm` (asumido como existente en el desarrollo), que establece que dos sustituciones sobre variables distintas conmutan siempre que se satisfagan ciertas condiciones de no ocurrencia:

1. La variable v no debe ocurrir en $u.\text{val}$, lo cual se cumple porque u es un término cerrado (`NotOccur_of_isClosed`).

2. La variable w no debe ocurrir en $t.\text{val}$, análogamente por ser t cerrado.

3. La variable w no debe ocurrir en $u.\text{val}$, también por clausura de u.

Significado para la composición de fórmulas

Este lema de conmutación es particularmente útil cuando se trabaja con `compose_formula`. En la verificación de la condición de gráfico para la composición, aparecen expresiones con múltiples sustituciones encadenadas sobre variables distintas (por ejemplo, v_0, v_1, y la variable ligada w). La capacidad de reordenar estas sustituciones sin alterar el resultado permite simplificar las expresiones y reducirlas a formas canónicas donde los lemas de representabilidad de φ_f y φ_g pueden aplicarse directamente. La clausura de los términos involucrados (son códigos de Gödel, que son numerales cerrados) garantiza que las condiciones de no ocurrencia se satisfacen automáticamente, haciendo que la conmutación sea aplicable de manera uniforme.

```
lemma substForm_substForm_term_comm {s : Nat} (φ : ArithForm s) (v w : Nat) (u : ArithTerm s) (hw : NotOccurForm w φ) :
  substForm w u (substForm v (ArithTerm.var w) φ) = substForm v u φ :=
 match φ with
 | ArithForm.bottom => rfl
 | ArithForm.eq t1 t2 => by
  have hw_left : NotOccur w t1 := hw.left
  have hw_right : NotOccur w t2 := hw.right
  rw [substForm_eq, substForm_eq]
  rw [substTerm_substTerm_term_comm t1 v w u hw_left, substTerm_substTerm_term_comm t2 v w u hw_right]
  rfl
 | ArithForm.impl f1 f2 => by
  have hw_left : NotOccurForm w f1 := hw.left
  have hw_right : NotOccurForm w f2 := hw.right
  rw [substForm_impl, substForm_impl]
  rw [substForm_substForm_term_comm f1 v w u hw_left, substForm_substForm_term_comm f2 v w u hw_right]
  rfl
 | ArithForm.neg f => by
  have hw_inner : NotOccurForm w f := hw
  rw [substForm_neg, substForm_neg]
  rw [substForm_substForm_term_comm f v w u hw_inner]
  rfl
 | ArithForm.forall_form n f => by
  have hw_left : w ≠ n := hw.left
  have hw_right : NotOccurForm w f := hw.right
  by_cases h_v : v = n
```

```
    · subst h_v
      rw [substForm_forall_form, if_pos rfl]
      rw [substForm_forall_form, if_neg hw_left]
      rw [substForm_fresh f w u hw_right]
      rw [substForm_forall_form, if_pos rfl]
    · rw [substForm_forall_form, if_neg h_v]
      rw [substForm_forall_form, if_neg hw_left]
      rw [substForm_substForm_term_comm f v w u hw_right]
      rw [substForm_forall_form, if_neg h_v]
  | ArithForm.provable_op heq t => by
    have hw_inner : NotOccur w t := hw
    rw [substForm_provable_op, substForm_provable_op]
    rw [substTerm_substTerm_term_comm t v w u hw_inner]
    rfl
  | ArithForm.true_op heq t => by
    have hw_inner : NotOccur w t := hw
    rw [substForm_true_op, substForm_true_op]
    rw [substTerm_substTerm_term_comm t v w u hw_inner]
    rfl
  | ArithForm.list_nth_eq t1 t2 t3 => by
    have hw_t1 : NotOccur w t1 := hw.left
    have hw_t2 : NotOccur w t2 := hw.right.left
    have hw_t3 : NotOccur w t3 := hw.right.right
    rw [substForm_list_nth_eq, substForm_list_nth_eq]
    rw [substTerm_substTerm_term_comm t1 v w u hw_t1, substTerm_substTerm_term_comm t2 v w u hw_t2, substTerm_substTerm_term_comm t3 v w u hw_t3]
    rfl
```

```
lemma subst_comm_closed {s : Nat} (φ : ArithForm s) (v w : Nat) (t u : TermClosed s)
(h_ne : v ≠ w) :

  substForm v t.val (substForm w u.val φ) = substForm w u.val (substForm v t.val φ)
:= by

 rw [substForm_comm φ v w t.val u.val h_ne]

 · rw [substTerm_fresh u.val v t.val (NotOccur_of_isClosed u.val u.property v)]

 · exact NotOccur_of_isClosed t.val t.property w

 · exact NotOccur_of_isClosed u.val u.property v
```

Lema `subst_compose`: Simplificación de la sustitución en la fórmula de composición

Se demuestra un lema técnico fundamental que simplifica la aplicación de sustituciones a la fórmula de composición `compose_formula`. Cuando se instancian las variables libres v_0 y v_1 con un numeral $\overline{x}$ y un término cerrado Y respectivamente, la fórmula resultante adopta una forma canónica que revela la semántica composicional:

$$ \forall \varphi_f, \varphi_g \in \text{ArithForm } } s,\ \forall x \in \mathbb{N},\ \forall Y \in \text{ArithTerm } } s \text{ cerrado},\ \forall w \notin \{0, 1\}, $$

$$ \text{compose_formula}(\varphi_f, \varphi_g, w)[v_0 := \overline{x},\ v_1 := Y] = $$

$$ \exists v_w\ (\varphi_g[v_0 := \overline{x},\ v_1 := v_w] \land \varphi_f[v_0 := v_w,\ v_1 := Y]) $$

Estructura de la demostración

La prueba consiste en una serie de reescrituras que expanden las definiciones y aplican lemas de conmutación de sustituciones. El proceso se desarrolla en varias etapas:

1. Expansión de cuantificadores: Se aplican los lemas `substForm_exists_form_fresh` para que las sustituciones de v_0 y v_1 atraviesen el cuantificador existencial $\exists v_w$, lo cual es posible porque $w \neq 0$ y $w \neq 1$ (las variables de sustitución son distintas de la variable ligada).

2. Conmutación de sustituciones en φ_g: Se intercambia el orden de las sustituciones sobre φ_g utilizando `substForm_comm`. Inicialmente se tiene $\text{substForm}(0, \overline{x}, \text{substForm}(1, \text{var } w, \varphi_g))$. Dado que $\overline{x}$ es un numeral cerrado (no contiene a v_1) y $\text{var } w$ no contiene a v_0 (pues $w \neq 0$), las condiciones de no ocurrencia se satisfacen y las sustituciones conmutan, produciendo $\text{substForm}(1, \text{var } w, \text{substForm}(0, \overline{x}, \varphi_g))$.

3. Simplificación de sustitución redundante en φ_f: La sustitución $\text{substForm}(0, \overline{x}, \text{substForm}(0, \text{var } w, \varphi_f))$ se simplifica porque dos sustituciones consecutivas sobre la misma variable v_0 colapsan en una sola: la sustitución más interna ($v_0 := \text{var } w$) es inmediatamente sobrescrita por la más externa ($v_0 := \overline{x}$). El lema `substForm_substForm_same` junto con `substTerm_var_fresh` (ya que $w \neq 0$) reducen esto a simplemente $\text{substForm}(0, \text{var } w, \varphi_f)$.

4. Simplificación de sustitución redundante en la φ_g ya conmutada: Análogamente, la sustitución $\text{substForm}(1, Y, \text{substForm}(1, \text{var } w, \dots))$ colapsa porque $w \neq 1$, dejando solo la sustitución interna.

5. Conmutación final en φ_f: Se intercambian las sustituciones $\text{substForm}(1, Y, \text{substForm}(0, \text{var } w, \varphi_f))$ utilizando nuevamente `substForm_comm`. Dado que Y es cerrado (no contiene a v_0) y $\text{var } w$ no contiene a v_1 (pues $w \neq 1$), las condiciones se satisfacen y

el resultado es $\text{substForm}(0, \text{var } w, \text{substForm}(1, Y, \varphi_f))$.

Significado semántico

El resultado final tiene una interpretación semántica transparente: la composición aplicada a la entrada $\overline{x}$ y con salida Y es equivalente a afirmar que existe un valor intermedio (representado por la variable w) tal que:

- φ_g relaciona la entrada $\overline{x}$ con el valor intermedio w (es decir, $w = g(x)$ internamente),

- φ_f relaciona el valor intermedio w con la salida Y (es decir, $Y = f(w)$ internamente).

Esta descomposición es exactamente la semántica pretendida de la composición de funciones y valida que `compose_formula` captura correctamente la noción de composición en el lenguaje objeto. La demostración pone de manifiesto la potencia de los lemas de conmutación de sustituciones, que permiten manipular expresiones complejas con múltiples cuantificadores y sustituciones de manera algebraica, reduciéndolas a formas canónicas donde las hipótesis de representabilidad de φ_f y φ_g pueden aplicarse directamente.

lemma subst_compose {s : Nat} (φ_f φ_g : ArithForm s) (x : Nat) (Y : ArithTerm s) (h_closed_Y : isClosed Y = true) (w : Nat) (hw0 : w ≠ 0) (hw1 : w ≠ 1) :

  substForm 1 Y (substForm 0 (natToTerm x) (compose_formula φ_f φ_g w)) =

```
  exists_form w (and_form (substForm 1 (ArithTerm.var w) (substForm 0 (natToTerm x) φ_g)) (substForm 0 (ArithTerm.var w) (substForm 1 Y φ_f))) := by
  unfold compose_formula
  rw [substForm_exists_form_fresh 0 (natToTerm x) w _ hw0]
  rw [substForm_exists_form_fresh 1 Y w _ hw1]
  congr 1
  dsimp [and_form, substForm, substTerm]

  have h_comm_g : substForm 0 (natToTerm x) (substForm 1 (ArithTerm.var w) φ_g) =
        substForm 1 (ArithTerm.var w) (substForm 0 (natToTerm x) φ_g) := by
    have h_closed := isClosed_natToTerm (s := s) x
    have h_not_occur : NotOccur 1 (natToTerm x) := NotOccur_of_isClosed (natToTerm x) h_closed 1
    have h_ne : 0 ≠ w := by omega
    have h_not_occur_vu : NotOccur (s := s) 0 (ArithTerm.var (s := s) w) := NotOccur.var (n := 0) (s := s) w h_ne
    have h_comm := substForm_comm φ_g 0 1 (natToTerm x) (ArithTerm.var w) (by decide) h_not_occur h_not_occur_vu
    have h_substTerm : substTerm 0 (natToTerm x) (ArithTerm.var (s := s) w) = ArithTerm.var (s := s) w := substTerm_var_fresh 0 (natToTerm x) w hw0
    rw [h_substTerm] at h_comm
    exact h_comm

  have h_same_f : substForm 0 (natToTerm x) (substForm 0 (ArithTerm.var w) φ_f) =
        substForm 0 (ArithTerm.var w) φ_f := by
    rw [← substForm_substForm_same φ_f 0 (natToTerm x) (ArithTerm.var w)]
    rw [substTerm_var_fresh 0 (natToTerm x) w hw0]

  rw [h_comm_g, h_same_f]
```

```
 have h_same_g2 : substForm 1 Y (substForm 1 (ArithTerm.var w) (substForm 0 (natToTerm x) φ_g)) =
            substForm 1 (ArithTerm.var w) (substForm 0 (natToTerm x) φ_g) := by
  rw [← substForm_substForm_same (substForm 0 (natToTerm x) φ_g) 1 Y (ArithTerm.var w)]
  rw [substTerm_var_fresh 1 Y w hw1]

 have h_comm_f2 : substForm 1 Y (substForm 0 (ArithTerm.var w) φ_f) =
            substForm 0 (ArithTerm.var w) (substForm 1 Y φ_f) := by
  have h_not_occur : NotOccur 0 Y := NotOccur_of_isClosed Y h_closed_Y 0
  have h_ne : 1 ≠ w := by omega
  have h_not_occur_vu : NotOccur (s := s) 1 (ArithTerm.var (s := s) w) := NotOccur.var (n := 1) (s := s) w h_ne
  have h_comm := substForm_comm φ_f 1 0 Y (ArithTerm.var w) (by decide) h_not_occur h_not_occur_vu
  have h_substTerm : substTerm 1 Y (ArithTerm.var (s := s) w) = ArithTerm.var (s := s) w := substTerm_var_fresh 1 Y w hw1
  rw [h_substTerm] at h_comm
  exact h_comm

 rw [h_same_g2, h_comm_f2]
```

Teorema principal `compose_representable`: Representabilidad de la composición de funciones

Se demuestra el teorema central sobre la clausura de las funciones representables bajo composición. Si dos funciones $f, g : \mathbb{N} \to \mathbb{N}$ son representables en Δ mediante fórmulas φ_f y φ_g respectivamente, entonces la función compuesta $f \circ g$ también es representable, mediante la fórmula `compose_formula`:

$$ \forall f, g : \mathbb{N} \to \mathbb{N},\ \forall \varphi_f, \varphi_g \in \text{ArithForm } s, $$

si φ_f representa a f y φ_g representa a g, entonces $\text{compose_formula}(\varphi_f, \varphi_g, w)$ representa a $f \circ g$, siempre que w sea una variable fresca (distinta de 0 y 1, y que no ocurra libre en φ_f ni en φ_g).

Hipótesis del teorema

Las hipótesis `repr_f` y `repr_g` establecen que para cada $x \in \mathbb{N}$:

- **Gráfico:** $\Delta \vdash \varphi_f(\overline{x}, \overline{f(x)})$ y $\Delta \vdash \varphi_g(\overline{x}, \overline{g(x)})$

- **Funcionalidad:** Si $\Delta \vdash \varphi_f(\overline{x}, Y)$ entonces $\Delta \vdash Y = \overline{f(x)}$, y análogamente para g.

Las condiciones de frescura garantizan que w no interfiere con las variables libres de φ_f y φ_g, lo cual es necesario para que las sustituciones conmuten correctamente.

Estructura de la demostración

La prueba se divide en dos partes, correspondientes a las dos condiciones de representabilidad.

Parte 1: Condición de gráfico

Se debe demostrar $\Delta \vdash \text{compose_formula}(\varphi_f, \varphi_g, w)(\overline{x}, \overline{f(g(x))})$.

1. Simplificación de la sustitución: Se aplica el lema `subst_compose` para reescribir la fórmula instanciada como:

$$ \exists v_w\ (\varphi_g[v_0 := \overline{x},\ v_1 := v_w] \land \varphi_f[v_0 := v_w,\ v_1 := \overline{f(g(x))}]) $$

2. Elección del testigo: Se toma como testigo el numeral $\overline{g(x)}$. Se debe demostrar que la instancia del cuerpo de la fórmula con $v_w := \overline{g(x)}$ pertenece a Δ.

3. Simplificación del cuerpo instanciado: Se aplican los lemas `substForm_substForm_term_comm` (dos veces, una para φ_g y otra para φ_f) para simplificar las sustituciones encadenadas. La condición de frescura $w \notin \text{FV}(\varphi_g)$ garantiza que la sustitución $v_1 := \text{var } w$ seguida de $w := \overline{g(x)}$ colapsa en $v_1 := \overline{g(x)}$. El resultado es:

$$ \varphi_g[v_0 := \overline{x},\ v_1 := \overline{g(x)}] \land \varphi_f[v_0 := \overline{g(x)},\ v_1 := \overline{f(g(x))}] $$

4. Conmutación en φ_f: Se aplica `subst_comm_closed` para intercambiar las sustituciones sobre φ_f, transformando $\varphi_f[v_0 := \overline{g(x)},\ v_1 := \overline{f(g(x))}]$ en una forma que coincide exactamente con la hipótesis `repr_f (g x)`.

5. Clausura deductiva: Por `repr_g x`, la primera componente de la conjunción pertenece a Δ. Por `repr_f (g x)` y la conmutación del paso anterior, la segunda componente también pertenece a Δ. Mediante `and_intro` se obtiene la conjunción, y mediante `exists_intro` se introduce el cuantificador existencial con testigo $\overline{g(x)}$.

Parte 2: Condición de funcionalidad

Se asume $\Delta \vdash \text{compose_formula}(\varphi_f, \varphi_g, w)(\overline{x}, Y)$ para algún término cerrado Y, y se debe demostrar $\Delta \vdash Y = \overline{f(g(x))}$.

1. Simplificación de la hipótesis: Se aplica `subst_compose` para reescribir la hipótesis como:

$$ \exists v_w\ (\varphi_g[v_0 := \overline{x},\ v_1 := v_w] \land \varphi_f[v_0 := v_w,\ v_1 := Y]) \in \Delta $$

2. Extracción del testigo: Por la propiedad de Henkin, existe un término cerrado Z tal que la instancia del cuerpo con $v_w := Z$ pertenece a Δ.

3. Simplificación del cuerpo instanciado: Análogamente al caso del gráfico, se aplican los lemas de conmutación para colapsar las sustituciones encadenadas, obteniendo:

$$ \varphi_g[v_0 := \overline{x},\ v_1 := Z] \land \varphi_f[v_0 := Z,\ v_1 := Y] \in \Delta $$

4. Determinación de Z: Por `and_elim_left_Δ`, se tiene $\varphi_g(\overline{x}, Z) \in \Delta$. Por la funcionalidad de φ_g (`repr_g x`), se deduce $Z = \overline{g(x)} \in \Delta$.

5. Sustitución de Z por $\overline{g(x)}$ en φ_f: Utilizando la regla de sustitución de iguales en el sistema deductivo (`eq_subst`), se reemplaza Z por $\overline{g(x)}$ en la segunda componente de la conjunción, obteniendo $\varphi_f[v_0 := \overline{g(x)},\ v_1 := Y] \in \Delta$.

6. Conmutación y determinación de Y: Se aplica `subst_comm_closed` para intercambiar las sustituciones en φ_f, y por la funcionalidad de φ_f (`repr_f (g x)`), se concluye $Y = \overline{f(g(x))} = \overline{(f \circ g)(x)} \in \Delta$.

Significado del teorema

`compose_representable` es un resultado fundamental en la teoría de la representabilidad: establece que la clase de funciones representables en Δ es cerrada bajo composición. Este es uno de los pasos clave en la demostración de que todas las funciones recursivas primitivas son representables, siguiendo la estrategia clásica de Gödel.

La demostración ilustra de manera ejemplar la interacción entre los distintos componentes de la infraestructura desarrollada:

- **Lemas de conmutación de sustituciones** (`substForm_substForm_term_comm`, `subst_comm_closed`) para manipular fórmulas con múltiples cuantificadores.

- **Propiedad de Henkin** para extraer testigos de fórmulas existenciales.

- **Reglas lógicas en Δ** (`and_elim_left_Δ`, `and_intro`, etc.) para descomponer y recomponer demostraciones.

- **Regla de sustitución de iguales** (`eq_subst`) para reemplazar términos por sus iguales demostrables.

La estructura modular de la prueba —reducción de la composición a la conjunción de las fórmulas representadoras individuales, y uso de la funcionalidad de cada una para determinar los valores intermedios— es un patrón que se repetirá en la demostración de clausura bajo otras operaciones (como recursión primitiva), constituyendo el núcleo de la aritmetización de la sintaxis.

```
--
============================================================================

-- DEMOSTRACIÓN TEOREMA PRINCIPAL: compose_representable

--
============================================================================


lemma compose_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

  (f g : Nat → Nat)

  (φ_f φ_g : ArithForm s)

  (w : Nat)

  (hw0 : w ≠ 0)

  (hw1 : w ≠ 1)

  (h_fresh_f : NotOccurForm w φ_f)

  (h_fresh_g : NotOccurForm w φ_g)

  (repr_f : ∀ (x : Nat),
```

```
    (substForm 1 (natToTerm (f x)) (substForm 0 (natToTerm x) φ_f) ∈ Δ) ∧
    (∀ (Y : TermClosed s),
     substForm 1 Y.val (substForm 0 (natToTerm x) φ_f) ∈ Δ →
     ArithForm.eq Y.val (natToTerm (f x)) ∈ Δ))
  (repr_g : ∀ (x : Nat),
    (substForm 1 (natToTerm (g x)) (substForm 0 (natToTerm x) φ_g) ∈ Δ) ∧
    (∀ (Y : TermClosed s),
     substForm 1 Y.val (substForm 0 (natToTerm x) φ_g) ∈ Δ →
     ArithForm.eq Y.val (natToTerm (g x)) ∈ Δ))
  (x : Nat) :
  let φ_fg := compose_formula φ_f φ_g w
  (substForm 1 (natToTerm ((f ∘ g) x)) (substForm 0 (natToTerm x) φ_fg) ∈ Δ) ∧
  (∀ (Y : TermClosed s),
    substForm 1 Y.val (substForm 0 (natToTerm x) φ_fg) ∈ Δ →
    ArithForm.eq Y.val (natToTerm ((f ∘ g) x)) ∈ Δ) := by
 let φ_fg := compose_formula φ_f φ_g w
 constructor
 · -- Gráfico
   rw [subst_compose φ_f φ_g x (natToTerm ((f ∘ g) x)) (isClosed_natToTerm ((f ∘ g)
x)) w hw0 hw1]
   let t_g : ArithTerm s := @natToTerm s (g x)
   have h_closed_tg := isClosed_natToTerm (s := s) (g x)
   let tg_closed : TermClosed s := ⟨t_g, h_closed_tg⟩
   let body := and_form (substForm 1 (ArithTerm.var w) (substForm 0 (natToTerm x)
φ_g)) (substForm 0 (ArithTerm.var w) (substForm 1 (natToTerm ((f ∘ g) x)) φ_f))
   have h_free : IsFreeFor w t_g body := IsFreeFor_closed body w tg_closed


   have h_subst_body : substForm w t_g body =
```

```
                and_form (substForm 1 t_g (substForm 0 (natToTerm x) φ_g))
(substForm 0 t_g (substForm 1 (natToTerm ((f ∘ g) x)) φ_f)) := by

    dsimp [body, and_form, substForm, substTerm]

    have h_comm_g : substForm w t_g (substForm 1 (ArithTerm.var w) (substForm 0
(natToTerm x) φ_g)) =

            substForm 1 t_g (substForm 0 (natToTerm x) φ_g) := by

      have h_fresh_g_subst : NotOccurForm w (substForm 0 (natToTerm x) φ_g) := by

        apply NotOccurForm_subst_other

        · exact h_fresh_g

        · exact NotOccur_of_isClosed (natToTerm x) (isClosed_natToTerm x) w

      exact substForm_substForm_term_comm (substForm 0 (natToTerm x) φ_g) 1 w
t_g h_fresh_g_subst

    have h_comm_f : substForm w t_g (substForm 0 (ArithTerm.var w) (substForm 1
(natToTerm (f (g x))) φ_f)) =

            substForm 0 t_g (substForm 1 (natToTerm (f (g x))) φ_f) := by

      have h_fresh_f_subst : NotOccurForm w (substForm 1 (natToTerm (f (g x))) φ_f) :=
by

        apply NotOccurForm_subst_other

        · exact h_fresh_f

        · exact NotOccur_of_isClosed (natToTerm (f (g x))) (isClosed_natToTerm (f (g x)))
w

      exact substForm_substForm_term_comm (substForm 1 (natToTerm (f (g x))) φ_f)
0 w t_g h_fresh_f_subst

    rw [h_comm_g, h_comm_f]


   have h_g_in := (repr_g x).left

   have h_f_in := (repr_f (g x)).left

   have h_f_comm : substForm 0 t_g (substForm 1 (natToTerm ((f ∘ g) x)) φ_f) =

            substForm 1 (natToTerm (f (g x))) (substForm 0 (natToTerm (g x)) φ_f) :=
by

    change substForm 0 tg_closed.val (substForm 1 (natToTerm (f (g x))) φ_f) =
```

```
        substForm 1 (natToTerm (f (g x))) (substForm 0 tg_closed.val φ_f)

    have h_comm_closed := subst_comm_closed φ_f 0 1 tg_closed ⟨natToTerm (f (g x)),
isClosed_natToTerm (f (g x))⟩ (by decide)

    exact h_comm_closed


  have h_f_in_comm : substForm 0 t_g (substForm 1 (natToTerm ((f ∘ g) x)) φ_f) ∈ Δ :=
by

    rwa [h_f_comm]


  have h_and_ded := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_g_in)
(SPA_Deduction_Internal.hyp Δ _ h_f_in_comm)

  have h_and_in := maximal_consistent_closed_under_deduction Δ h_max h_con _
h_and_ded


  have h_exists_ded := exists_intro Δ w body t_g h_free (SPA_Deduction_Internal.hyp Δ
_ (by rwa [h_subst_body]))

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists_ded


· -- Funcionalidad

  intro Y h_in

  rw [subst_compose φ_f φ_g x Y.val Y.property w hw0 hw1] at h_in

  let body := and_form (substForm 1 (ArithTerm.var w) (substForm 0 (natToTerm x)
φ_g)) (substForm 0 (ArithTerm.var w) (substForm 1 Y.val φ_f))

  obtain ⟨Z, hZ⟩ := h_henkin w body h_in


  have h_subst_body : substForm w Z.val body =

            and_form (substForm 1 Z.val (substForm 0 (natToTerm x) φ_g))
(substForm 0 Z.val (substForm 1 Y.val φ_f)) := by

    dsimp [body, and_form, substForm, substTerm]

    have h_comm_g : substForm w Z.val (substForm 1 (ArithTerm.var w) (substForm 0
(natToTerm x) φ_g)) =
```

```
            substForm 1 Z.val (substForm 0 (natToTerm x) φ_g) := by
      have h_fresh_g_subst : NotOccurForm w (substForm 0 (natToTerm x) φ_g) := by
        apply NotOccurForm_subst_other
        · exact h_fresh_g
        · exact NotOccur_of_isClosed (natToTerm x) (isClosed_natToTerm x) w
      exact substForm_substForm_term_comm (substForm 0 (natToTerm x) φ_g) 1 w
Z.val h_fresh_g_subst
    have h_comm_f : substForm w Z.val (substForm 0 (ArithTerm.var w) (substForm 1
Y.val φ_f)) =
            substForm 0 Z.val (substForm 1 Y.val φ_f) := by
      have h_fresh_f_subst : NotOccurForm w (substForm 1 Y.val φ_f) := by
        apply NotOccurForm_subst_other
        · exact h_fresh_f
        · exact NotOccur_of_isClosed Y.val Y.property w
      exact substForm_substForm_term_comm (substForm 1 Y.val φ_f) 0 w Z.val
h_fresh_f_subst
    rw [h_comm_g, h_comm_f]
  rw [h_subst_body] at hZ

  have h_g_Z := and_elim_left_Δ h_max h_con _ _ hZ
  have h_f_Z := and_elim_right_Δ h_max h_con _ _ hZ
  have h_eq_Z := (repr_g x).right Z h_g_Z

  have h_f_Z_ded := SPA_Deduction_Internal.hyp Δ _ h_f_Z
  have h_eq_Z_ded := SPA_Deduction_Internal.hyp Δ _ h_eq_Z
  let t_g := @natToTerm s (g x)
  have h_closed_tg := isClosed_natToTerm (s := s) (g x)
  let tg_closed : TermClosed s := ⟨t_g, h_closed_tg⟩
```

```
  have h_free1 : IsFreeFor 0 Z.val (substForm 1 Y.val φ_f) := IsFreeFor_closed _ 0 Z

  have h_free2 : IsFreeFor 0 t_g (substForm 1 Y.val φ_f) := IsFreeFor_closed _ 0 tg_closed


  have h_subst_ded := SPA_Deduction_Internal.eq_subst Δ 0 (substForm 1 Y.val φ_f) Z.val t_g h_eq_Z_ded h_f_Z_ded h_free1 h_free2

  have h_subst_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_subst_ded


  have h_f_comm : substForm 0 t_g (substForm 1 Y.val φ_f) =

            substForm 1 Y.val (substForm 0 t_g φ_f) := by

    change substForm 0 tg_closed.val (substForm 1 Y.val φ_f) =

      substForm 1 Y.val (substForm 0 tg_closed.val φ_f)

    have h_comm_closed := subst_comm_closed φ_f 0 1 tg_closed Y (by decide)

    exact h_comm_closed

  rw [h_f_comm] at h_subst_in


  have h_final := (repr_f (g x)).right Y h_subst_in

  exact h_final
```

Lemas auxiliares: Asociatividad de la adición con numerales y fórmula de orden con variable testigo arbitraria

Se demuestran dos lemas que extienden la infraestructura aritmética y de orden dentro de Δ: la asociatividad de la adición cuando el tercer sumando es un numeral, y la capacidad de introducir la fórmula de orden con cualquier variable testigo.

Lema `add_assoc_numeral_Δ`: Asociatividad de la adición con numeral

Establece que la adición es asociativa en Δ cuando el tercer término es un numeral $\overline{n}$:

$$ \forall u, v \in \text{ArithTerm } s,\ \forall n \in \mathbb{N},\ \Delta \vdash (u + v) + \overline{n} = u + (v + \overline{n}) $$

Estructura de la demostración

La prueba procede por inducción sobre n:

Caso base ($n = 0$): Se debe demostrar $(u + v) + 0 = u + (v + 0)$. Por el axioma `add_zero_axiom` aplicado a $(u + v)$, se tiene $(u + v) + 0 = u + v \in \Delta$. Por el mismo axioma aplicado a v, se tiene $v + 0 = v \in \Delta$. Por congruencia de la adición con u, se obtiene $u + (v + 0) = u + v \in \Delta$. Por transitividad, ambas expresiones son iguales a $u + v$, y por tanto iguales entre sí.

Paso inductivo ($n = k + 1$): Se asume la hipótesis de inducción $(u + v) + \overline{k} = u + (v + \overline{k}) \in \Delta$.

1. Por el axioma `add_succ_axiom` para $(u + v)$ y $\overline{k}$: $(u + v) + S(\overline{k}) = S((u + v) + \overline{k}) \in \Delta$.

2. Por hipótesis de inducción y congruencia del sucesor: $S((u + v) + \overline{k}) = S(u + (v + \overline{k})) \in \Delta$.

3. Por transitividad con el paso 1: $(u + v) + S(\overline{k}) = S(u + (v + \overline{k})) \in \Delta$.

4. Por el axioma `add_succ_axiom` para v y $\overline{k}$: $v + S(\overline{k}) = S(v + \overline{k}) \in \Delta$.

5. Por congruencia de la adición con u: $u + (v + S(\overline{k})) = u + S(v + \overline{k}) \in \Delta$.

6. Por el axioma `add_succ_axiom` para u y $(v + \overline{k})$: $u + S(v + \overline{k}) = S(u + (v + \overline{k})) \in \Delta$.

7. Por transitividad de los pasos 4-6 y simetría de la igualdad: $S(u + (v + \overline{k})) = u + (v + S(\overline{k})) \in \Delta$.

8. Finalmente, por transitividad con el paso 3, se obtiene $(u + v) + S(\overline{k}) = u + (v + S(\overline{k})) \in \Delta$.

Lema `eq_mul_Δ`: Congruencia de la multiplicación en Δ

$$ \forall t_1, t_2, u_1, u_2 \in \text{ArithTerm } s,\ t_1 = t_2 \in \Delta \land u_1 = u_2 \in \Delta \implies t_1 \cdot u_1 = t_2 \cdot u_2 \in \Delta $$

Análogo a `eq_add_Δ` pero para la multiplicación. La demostración invoca la regla `eq_mul` del sistema deductivo y cierra por maximalidad.

Lema `lt_form_intro_w`: Introducción de la fórmula de orden con variable testigo arbitraria

Generaliza `lt_form_intro_of_lt` permitiendo especificar la variable testigo w del cuantificador existencial, en lugar de usar una variable fija:

$$ \forall a, b \in \mathbb{N},\ a < b \implies \forall w \in \mathbb{N},\ \Delta \vdash \overline{a} < \overline{b} \text{ (con variable testigo } w \text{)} $$

Estructura de la demostración

La prueba es una adaptación directa de `lt_form_intro_of_lt`, reemplazando la variable testigo 3 por el parámetro w:

1. Se calcula $k = b - a - 1$ tal que $b = a + (k + 1)$.

2. Por el homomorfismo de la adición, $\overline{b} = \overline{a} + \overline{k + 1} = \overline{a} + S(\overline{k}) \in \Delta$.

3. Se sustituye la variable w por $\overline{k}$ en la fórmula $\overline{b} = \overline{a} + S(v_w)$. La simplificación es directa porque $\overline{b}$ y $\overline{a}$ son cerrados y la variable w solo aparece como $S(v_w)$, que se convierte en $S(\overline{k})$.

4. Se aplica introducción del existencial con testigo $\overline{k}$ para obtener $\exists v_w\ (\overline{b} = \overline{a} + S(v_w)) \in \Delta$, que es exactamente $\overline{a} < \overline{b}$ con variable testigo w.

Significado

Esta generalización es técnicamente necesaria cuando se trabaja con `compose_formula` u otras construcciones que introducen variables ligadas con índices específicos. La capacidad de elegir la variable testigo permite evitar colisiones con otras variables ligadas en el contexto, garantizando que las fórmulas de orden pueden insertarse en cualquier entorno sintáctico sin riesgo de captura.

/-- 1. Lema de asociatividad de la adición cuando el tercer término es un numeral natToTerm n -/

lemma add_assoc_numeral_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

```
  (u v : ArithTerm s) (n : Nat) :
  ArithForm.eq (ArithTerm.add (ArithTerm.add u v) (natToTerm n))
    (ArithTerm.add u (ArithTerm.add v (natToTerm n))) ∈ Δ := by
 induction n with
 | zero =>
  have h1 := h_add_zero (ArithTerm.add u v)
  have h2 := h_add_zero v
  have h_refl_u := eq_refl_in_Δ Δ h_max h_con u
  have h3 := eq_add_Δ h_max h_con h_refl_u h2
  have h3_symm := eq_symm_Δ h_max h_con h3
  exact eq_trans_Δ h_max h_con h1 h3_symm
 | succ k ih =>
  have h1 := h_add_succ (ArithTerm.add u v) (natToTerm k)
  have h_ih_succ := eq_succ_Δ h_max h_con ih
  have h2 := eq_trans_Δ h_max h_con h1 h_ih_succ
  have h3 := h_add_succ u (ArithTerm.add v (natToTerm k))
  have h4 := h_add_succ v (natToTerm k)
  have h_refl_u := eq_refl_in_Δ Δ h_max h_con u
  have h5 := eq_add_Δ h_max h_con h_refl_u h4
  have h_right := eq_trans_Δ h_max h_con h5 h3
  exact eq_trans_Δ h_max h_con h2 (eq_symm_Δ h_max h_con h_right)



lemma eq_mul_Δ {s : Nat} {Δ : Set (ArithForm s)} (h_max : Maximal Δ) (h_con :
Consistent Δ)
  {t1 t2 u1 u2 : ArithTerm s} (h1 : ArithForm.eq t1 t2 ∈ Δ) (h2 : ArithForm.eq u1 u2 ∈
Δ) :
  ArithForm.eq (ArithTerm.mul t1 u1) (ArithTerm.mul t2 u2) ∈ Δ :=
 maximal_consistent_closed_under_deduction Δ h_max h_con _
```

(SPA_Deduction_Internal.eq_mul Δ t1 t2 u1 u2 (SPA_Deduction_Internal.hyp Δ _ h1) (SPA_Deduction_Internal.hyp Δ _ h2))

/-- 2. Lema de acotación para lt_form con variable de testigo 'w' arbitraria -/

lemma lt_form_intro_w {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(a b : Nat) (h_lt : a < b) (w : Nat) :

lt_form (s := s) (@natToTerm s a) (@natToTerm s b) w ∈ Δ := by

let k := b - a - 1

have h_eq_b : b = a + (k + 1) := by omega

have h_hom := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ a (k + 1)

have h_b_eq : @natToTerm s b = @natToTerm s (a + (k + 1)) := by rw [h_eq_b]

rw [← h_b_eq] at h_hom

have h_subst : substForm w (@natToTerm s k) (ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var w)))) =

ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (@natToTerm s k))) := by

dsimp [substForm, substTerm]

have h_b' := @substTerm_fresh s (@natToTerm s b) w (@natToTerm s k) (NotOccur_of_isClosed (@natToTerm s b) (isClosed_natToTerm b) w)

have h_a' := @substTerm_fresh s (@natToTerm s a) w (@natToTerm s k) (NotOccur_of_isClosed (@natToTerm s a) (isClosed_natToTerm a) w)

have h_if : (if w = w then @natToTerm s k else ArithTerm.var w) = @natToTerm s k := if_pos rfl

rw [h_b', h_a', h_if]

have h_hom_rew : substForm w (@natToTerm s k) (ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var w)))) ∈ Δ := by

rwa [h_subst]

have h_free : IsFreeFor w (@natToTerm s k) (ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var w)))) :=

IsFreeFor_closed _ w ⟨@natToTerm s k, isClosed_natToTerm k⟩

have h_exists := exists_intro Δ w (ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var w)))) (@natToTerm s k) h_free (SPA_Deduction_Internal.hyp Δ _ h_hom_rew)

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists

Lemas auxiliares para la función beta de Gödel: Igualdad del módulo y orden con término compuesto

Se demuestran dos lemas técnicos necesarios para la representabilidad de la función beta de Gödel. El primero establece una igualdad algebraica entre un numeral y un término compuesto que representa el módulo $1 + (I+1) \cdot B$. El segundo permite introducir la fórmula de orden estricto cuando el lado derecho es dicho término compuesto.

Lema `M_term_eq_Δ`: Igualdad del término módulo con su numeral

Establece que el término compuesto $S((S(\overline{I}) \cdot \overline{B}))$ es demostrablemente igual al numeral $\overline{1 + (I+1) \cdot B}$:

$$ \forall I, B \in \mathbb{N},\ \Delta \vdash \overline{1 + (I + 1) \cdot B} = S(S(\overline{I}) \cdot \overline{B}) $$

Estructura de la demostración

1. **Reordenamiento aritmético:** Se observa que $1 + (I + 1) \cdot B = (I + 1) \cdot B + 1$, por conmutatividad de la adición en $\mathbb{N}$.

2. **Aplicación del homomorfismo de la multiplicación:** El lema `natToTerm_mul_hom` proporciona $\overline{(I + 1) \cdot B} = \overline{I + 1} \cdot \overline{B} \in \Delta$. Dado que $\overline{I + 1} = S(\overline{I})$ por definición de `natToTerm`, el lado derecho es exactamente $S(\overline{I}) \cdot \overline{B}$.

3. **Aplicación de congruencia del sucesor:** La igualdad del paso 2 se envuelve con el constructor `succ`, obteniendo $S(\overline{(I + 1) \cdot B}) = S(S(\overline{I}) \cdot \overline{B}) \in \Delta$.

4. **Identificación con el numeral original:** El lado izquierdo $S(\overline{(I + 1) \cdot B})$ es precisamente $\overline{(I + 1) \cdot B + 1}$, que por el paso 1 coincide con $\overline{1 + (I + 1) \cdot B}$.

Este lema conecta la representación interna del módulo (como término compuesto) con su representación canónica como numeral, permitiendo alternar entre ambas según convenga a la demostración.

Lema `lt_form_M_term_intro`: Orden estricto con término módulo

Establece que si $R < 1 + (I + 1) \cdot B$ en el metalenguaje, entonces la fórmula de orden con el término compuesto $M_{\text{term}} = S(S(\overline{I}) \cdot \overline{B})$ como cota superior es demostrable en Δ:

$$ \forall R, I, B \in \mathbb{N},\ R < 1 + (I + 1) \cdot B \implies \Delta \vdash \overline{R} < M_{\text{term}} \text{ (con variable testigo 5)} $$

Estructura de la demostración

1. **Cálculo de la diferencia:** Sea $M = 1 + (I + 1) \cdot B$. Dado que $R < M$, existe $k = M - R - 1$ tal que $M = R + (k + 1)$.

2. **Cadena de igualdades:** Se construye una prueba en Δ de que $M_{\text{term}} = \overline{R} + S(\overline{k})$:

   - Por `M_term_eq_Δ`, se tiene $M_{\text{term}} = \overline{M}$ (módulo simetría).

   - Por la igualdad $M = R + (k + 1)$, se tiene $\overline{M} = \overline{R + (k + 1)}$.

   - Por el homomorfismo de la adición, $\overline{R + (k + 1)} = \overline{R} + \overline{k + 1} = \overline{R} + S(\overline{k})$.

   - Por transitividad, $M_{\text{term}} = \overline{R} + S(\overline{k}) \in \Delta$.

3. **Introducción del existencial:** Se sustituye la variable testigo 5 por $\overline{k}$ en la fórmula $M_{\text{term}} = \overline{R} + S(v_5)$. La simplificación es directa porque M_{term} y $\overline{R}$ son términos cerrados (no contienen a v_5), y la variable v_5 solo aparece en el sucesor. El resultado es precisamente la igualdad del paso 2.

4. **Cierre deductivo:** Se aplica `exists_intro` para obtener $\exists v_5\ (M_{\text{term}} = \overline{R} + S(v_5)) \in \Delta$, que es la definición de $\overline{R} < M_{\text{term}}$ con variable testigo 5.

Significado para la función beta de Gödel

Estos dos lemas son piezas preparatorias para la demostración de la representabilidad de la función beta de Gödel. La función beta se define como:

$$ \beta(a, b, i) = a \bmod (1 + (i + 1) \cdot b) $$

El término $M_{\text{term}} = S(S(\overline{I}) \cdot \overline{B})$ es precisamente la representación interna del módulo $1 + (I+1) \cdot B$. El lema `lt_form_M_term_intro` establece que el residuo R de la división es internamente menor que el módulo, una propiedad esencial para la corrección de la representación. La elección de la variable testigo 5 es arbitraria pero conveniente para evitar colisiones con otras variables ligadas en el contexto de la fórmula `BetaForm`, que utiliza las variables 0, 1, 2, 3 para los argumentos y el resultado, y las variables 4 y 5 como testigos existenciales auxiliares.

/-- 3. Igualdad en Δ para M_term y natToTerm M -/

lemma M_term_eq_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

```
  (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

  (I B : Nat) :

  ArithForm.eq (natToTerm (1 + (I + 1) * B))

   (ArithTerm.succ (ArithTerm.mul (ArithTerm.succ (@natToTerm s I)) (@natToTerm s B))) ∈ Δ := by

 have h_nat : 1 + (I + 1) * B = (I + 1) * B + 1 := by omega

 rw [h_nat]

 have h_mul := natToTerm_mul_hom Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ (I + 1) B

 exact eq_succ_Δ h_max h_con h_mul


/-- 4. Demostración directa de lt_form con M_term -/

lemma lt_form_M_term_intro {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

  (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

  (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

  (R I B : Nat) (h_lt : R < 1 + (I + 1) * B) :

  lt_form (s := s) (@natToTerm s R) (ArithTerm.succ (ArithTerm.mul (ArithTerm.succ (@natToTerm s I)) (@natToTerm s B))) 5 ∈ Δ := by

 let M := 1 + (I + 1) * B

 let k := M - R - 1

 have h_eq_M : M = R + (k + 1) := by omega

 let M_term : ArithTerm s := ArithTerm.succ (ArithTerm.mul (ArithTerm.succ (@natToTerm s I)) (@natToTerm s B))

 have h_M_eq := M_term_eq_Δ Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ I B

 have h_M_symm := eq_symm_Δ h_max h_con h_M_eq

 have h_add_hom := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ R (k + 1)
```

```
  have h_M_nat_eq : @natToTerm s M = @natToTerm s (R + (k + 1)) := by rw [h_eq_M]

  rw [h_M_nat_eq] at h_M_symm

  have h_M_body := eq_trans_Δ h_max h_con h_M_symm h_add_hom


  have h_subst : substForm 5 (@natToTerm s k) (ArithForm.eq M_term
(ArithTerm.add (@natToTerm s R) (ArithTerm.succ (ArithTerm.var 5)))) =

        ArithForm.eq M_term (ArithTerm.add (@natToTerm s R) (ArithTerm.succ
(@natToTerm s k))) := by

    dsimp [substForm, substTerm]

    have h_M' := @substTerm_fresh s M_term 5 (@natToTerm s k)
(NotOccur_of_isClosed M_term (by dsimp [M_term, isClosed]; rw
[isClosed_natToTerm, isClosed_natToTerm]; rfl) 5)

    have h_R' := @substTerm_fresh s (@natToTerm s R) 5 (@natToTerm s k)
(NotOccur_of_isClosed (@natToTerm s R) (isClosed_natToTerm R) 5)

    rw [h_M', h_R']


  have h_M_body_rew : substForm 5 (@natToTerm s k) (ArithForm.eq M_term
(ArithTerm.add (@natToTerm s R) (ArithTerm.succ (ArithTerm.var 5)))) ∈ Δ := by

    rwa [h_subst]


  have h_free : IsFreeFor 5 (@natToTerm s k) (ArithForm.eq M_term (ArithTerm.add
(@natToTerm s R) (ArithTerm.succ (ArithTerm.var 5)))) :=

    IsFreeFor_closed _ 5 ⟨@natToTerm s k, isClosed_natToTerm k⟩


  have h_ex := exists_intro Δ 5 _ (@natToTerm s k) h_free (SPA_Deduction_Internal.hyp
Δ _ h_M_body_rew)

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_ex
```

Lemas de simplificación de sustituciones sobre variables ligadas y fórmula auxiliar

Se establecen tres lemas técnicos sobre el comportamiento de las sustituciones cuando la variable de sustitución coincide con una variable ligada, y un lema de simplificación para una fórmula compuesta específica que aparecerá en la representabilidad de la función beta.

Lema `substForm_forall_same`: Sustitución sobre variable ligada

$$ \forall v \in \mathbb{N},\ \forall u \in \text{ArithTerm } s,\ \forall f \in \text{ArithForm } s,\ \text{substForm}(v, u, \forall v\ f) = \forall v\ f $$

Cuando la variable de sustitución v coincide con la variable ligada del cuantificador, la sustitución no tiene efecto: la variable está ligada y no debe ser reemplazada. La demostración es inmediata por la definición de `substForm`, que utiliza un condicional `if v = n` para decidir si atraviesa o no el cuantificador.

Lema `substTerm_var_same`: Sustitución de una variable por un término

$$ \forall v \in \mathbb{N},\ \forall u \in \text{ArithTerm } s,\ \text{substTerm}(v, u, \text{var } v) = u $$

La sustitución de la variable v por u en la variable v misma produce exactamente u. Es la propiedad fundamental de la sustitución.

Lema `IsFreeFor_forall_same`: Libertad de sustitución trivial

$$ \forall v \in \mathbb{N},\ \forall u \in \text{ArithTerm } s,\ \forall f \in \text{ArithForm } s,\ \text{IsFreeFor}(v, u, \forall v\ f) = \text{True} $$

La condición de "libre para sustituir" se satisface trivialmente cuando la variable de sustitución es la variable ligada, porque la sustitución no penetra el cuantificador y por tanto no hay riesgo de captura.

Lema `substForm_phi_prime`: Simplificación de una fórmula compuesta

Este lema simplifica la sustitución de una variable v por un término u en una fórmula φ' que codifica una propiedad inductiva de la adición. La fórmula φ' se define como:

$$ \varphi' := \text{Ax0}_{\text{succ}} \to (\text{Ax0}_x \to (\forall v\ \text{Ax1}_{\text{succ}} \to (\forall v\ \text{Ax1}_x \to \text{Eq}_v))) $$

donde:

- $\text{Ax0}_{\text{succ}} = \text{add_zero_axiom}(S(x))$: $(S(x) + 0 = S(x))$

- $\text{Ax0}_x = \text{add_zero_axiom}(x)$: $(x + 0 = x)$

- $\text{Ax1}_{\text{succ}} = \forall v\ \text{add_succ_axiom}(S(x), v)$: $\forall v\ (S(x) + S(v) = S(S(x) + v))$

- $\text{Ax1}_x = \forall v\ \text{add_succ_axiom}(x, v)$: $\forall v\ (x + S(v) = S(x + v))$

- $\text{Eq}_v = (S(x) + v = S(x + v))$

El lema establece que, bajo la hipótesis de que v no ocurre en x, la sustitución $\text{substForm}(v, u, \varphi')$ produce:

$$ \text{Ax0}_{\text{succ}} \to (\text{Ax0}_x \to (\forall v\ \text{Ax1}_{\text{succ}} \to (\forall v\ \text{Ax1}_x \to (S(x) + u = S(x + u))))) $$

Estructura de la demostración

1. **Preservación de x:** Dado que v no ocurre en x (por hipótesis `hv`), la sustitución de v por u en x no tiene efecto: `substTerm_fresh` garantiza $\text{substTerm}(v, u, x) = x$.

2. **Atravesamiento de los axiomas:** Los axiomas Ax0_{succ} y Ax0_x son igualdades entre términos que no contienen a v (porque x no contiene a v), por lo que la sustitución los deja inalterados.

3. **Cuantificadores sobre v:** Los axiomas $\forall v\ \text{Ax1}$ tienen a v como variable ligada, por lo que la sustitución no los atraviesa (por `substForm_forall_same`). La verificación explícita en la demostración utiliza que el condicional `if v = v` se evalúa a verdadero, tomando la rama que preserva el cuantificador.

4. **Fórmula Eq_v:** Esta es la única parte afectada por la sustitución. La variable v aparece libre en $S(x) + v = S(x + v)$, y la sustitución la reemplaza por u, produciendo $S(x) + u = S(x + u)$ (usando que x no contiene a v, por lo que las ocurrencias de x no son afectadas).

Significado para la representabilidad de la función beta

La fórmula φ' que aparece en este lema es parte de la construcción de la fórmula que representa la operación de módulo en la función beta de Gödel. Específicamente, φ' codifica la propiedad de que la adición es recursiva: saber que $x + 0 = x$ y que $x + S(v) = S(x + v)$ para todo v permite calcular $x + u$ para cualquier u por inducción. La simplificación `substForm_phi_prime` será utilizada cuando se instancie esta fórmula con un numeral concreto $\overline{k}$ en lugar de

u, permitiendo reducir la fórmula a una implicación entre instancias concretas de los axiomas y una igualdad específica sobre los numerales involucrados.

```
lemma substForm_forall_same {s : Nat} (v : Nat) (u : ArithTerm s) (f : ArithForm s) :
  substForm v u (ArithForm.forall_form v f) = ArithForm.forall_form v f := by
 dsimp [substForm]
 exact if_pos rfl

lemma substTerm_var_same {s : Nat} (v : Nat) (u : ArithTerm s) :
  substTerm v u (ArithTerm.var v) = u := by
 dsimp [substTerm]
 exact if_pos rfl

lemma IsFreeFor_forall_same {s : Nat} (v : Nat) (u : ArithTerm s) (f : ArithForm s) :
  IsFreeFor v u (ArithForm.forall_form v f) = True := by
 dsimp [IsFreeFor]
 exact if_pos rfl

lemma substForm_phi_prime {s : Nat} (x : ArithTerm s) (v : Nat) (hv : NotOccur v x) (u
: ArithTerm s) :
  let Ax0_succ : ArithForm s := add_zero_axiom (ArithTerm.succ x)
  let Ax0_x : ArithForm s := add_zero_axiom x
  let Ax1_succ_forall : ArithForm s := ArithForm.forall_form v (add_succ_axiom
(ArithTerm.succ x) (ArithTerm.var v))
  let Ax1_x_forall : ArithForm s := ArithForm.forall_form v (add_succ_axiom x
(ArithTerm.var v))
```

```
  let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.add (ArithTerm.succ x) (ArithTerm.var v)) (ArithTerm.succ (ArithTerm.add x (ArithTerm.var v)))

  let φ' : ArithForm s := ArithForm.impl Ax0_succ (ArithForm.impl Ax0_x (ArithForm.impl Ax1_succ_forall (ArithForm.impl Ax1_x_forall Eq_v)))

  substForm v u φ' =

  ArithForm.impl Ax0_succ (ArithForm.impl Ax0_x (ArithForm.impl Ax1_succ_forall (ArithForm.impl Ax1_x_forall (ArithForm.eq (ArithTerm.add (ArithTerm.succ x) u) (ArithTerm.succ (ArithTerm.add x u)))))) := by

  intro Ax0_succ Ax0_x Ax1_succ_forall Ax1_x_forall Eq_v φ'

  dsimp [φ', Ax0_succ, Ax0_x, Ax1_succ_forall, Ax1_x_forall, Eq_v, add_zero_axiom, add_succ_axiom, substForm, substTerm]

  have h_x : substTerm v u x = x := substTerm_fresh x v u hv

  rw [h_x]

  have h_vv : (if v = v then u else ArithTerm.var v) = u := if_pos rfl

  have h_if1 : (if v = v then ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.succ x) (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add (ArithTerm.succ x) (ArithTerm.var v)))) else ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.succ x) (ArithTerm.succ u)) (ArithTerm.succ (ArithTerm.add (ArithTerm.succ x) u)))) =

      ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.succ x) (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add (ArithTerm.succ x) (ArithTerm.var v)))) := if_pos rfl

  have h_if2 : (if v = v then ArithForm.forall_form v (ArithForm.eq (ArithTerm.add x (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add x (ArithTerm.var v)))) else ArithForm.forall_form v (ArithForm.eq (ArithTerm.add x (ArithTerm.succ u)) (ArithTerm.succ (ArithTerm.add x u)))) =

      ArithForm.forall_form v (ArithForm.eq (ArithTerm.add x (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add x (ArithTerm.var v)))) := if_pos rfl

  rw [h_vv, h_if1, h_if2]
```

Lema `succ_add_Δ`: Conmutatividad del sucesor con la adición en Δ

Se demuestra una propiedad algebraica fundamental de la aritmética dentro de Δ: el sucesor del primer sumando puede extraerse fuera de la adición. Formalmente:

$$ \forall x, y \in \text{ArithTerm } s,\ \Delta \vdash S(x) + y = S(x + y) $$

bajo las hipótesis de que Δ es maximal consistente, tiene la propiedad de Henkin, y contiene todas las instancias de los axiomas de la adición.

Estructura de la demostración

La prueba es notablemente elaborada y constituye una aplicación magistral del principio de inducción interna del sistema SPA. La estrategia consiste en demostrar la propiedad para todo y por inducción sobre y, pero utilizando una variable fresca v y el esquema de inducción del sistema deductivo.

1. Elección de variable fresca: Se define $v := \max_var_term(x) + 1$, garantizando que v no ocurre en x. Esta variable actuará como variable de inducción.

2. Construcción de la fórmula de inducción $\varphi'(v)$: Se define una fórmula compuesta que codifica la propiedad a demostrar:

$$ \varphi'(v) := \text{Ax0}_{S(x)} \to (\text{Ax0}_x \to (\forall v\ \text{Ax1}_{S(x)} \to (\forall v\ \text{Ax1}_x \to \text{Eq}_v))) $$

donde:

- $\text{Ax0}_{S(x)} = (S(x) + 0 = S(x))$

- $\text{Ax0}_x = (x + 0 = x)$

- $\text{Ax1}_{S(x)} = \forall v\ (S(x) + S(v) = S(S(x) + v))$

- $\text{Ax1}_x = \forall v\ (x + S(v) = S(x + v))$

- $\text{Eq}_v = (S(x) + v = S(x + v))$

La estructura de φ' es una implicación en cadena: si se tienen los axiomas de la adición para $S(x)$ y para x, entonces se cumple la igualdad deseada. Esta formulación permite aplicar inducción sobre v usando las reglas del sistema.

3. Caso base de la inducción ($v := 0$): Se demuestra $\varphi'(0)$, es decir, bajo los axiomas se tiene $S(x) + 0 = S(x + 0)$. La demostración usa:

- $\text{Ax0}_{S(x)}$ para obtener $S(x) + 0 = S(x)$

- Ax0_x para obtener $x + 0 = x$, de donde $S(x + 0) = S(x)$ por congruencia

- Transitividad para obtener la igualdad deseada.

4. Paso inductivo ($v \to S(v)$): Se demuestra $\forall v\ (\varphi'(v) \to \varphi'(S(v)))$. Asumiendo $\varphi'(v)$ y los axiomas, se debe probar $S(x) + S(v) = S(x + S(v))$. La cadena de igualdades es:

- Por $\text{Ax1}_{S(x)}$: $S(x) + S(v) = S(S(x) + v)$

- Por la hipótesis $\varphi'(v)$ (que da $S(x) + v = S(x + v)$) y congruencia: $S(S(x) + v) = S(S(x + v))$

- Por Ax1_x: $x + S(v) = S(x + v)$, de donde $S(x + S(v)) = S(S(x + v))$ por congruencia y simetría

- Por transitividad se obtiene $S(x) + S(v) = S(x + S(v))$.

5. Aplicación del esquema de inducción: El sistema SPA provee la regla `induction` que, a partir del caso base $\varphi'(0)$ y del paso inductivo $\forall v\ (\varphi'(v) \to \varphi'(S(v)))$, permite concluir $\forall v\ \varphi'(v)$. Esta conclusión se debilita a Δ.

6. Instanciación al término y: Se elimina el cuantificador universal sobre v instanciando con el término y, para lo cual se verifica que y es libre para v en φ' (lo cual es cierto porque v no aparece ligada de manera conflictiva en φ'). Se obtiene $\Delta \vdash \varphi'(y)$.

7. Eliminación de las hipótesis axiomáticas: La fórmula $\varphi'(y)$ es una implicación cuyas premisas son exactamente los axiomas de la adición. Dado que por hipótesis todos estos axiomas pertenecen a Δ (proporcionados por `h_add_zero` y `h_add_succ`), se aplica reiteradamente `imp_elim` para descargar cada premisa:

- Para `h_add_succ` se requiere un argumento adicional: si alguna instancia del axioma no estuviera en Δ, por maximalidad su negación estaría en Δ, y por la propiedad de Henkin se obtendría un contraejemplo que contradice la hipótesis de que todas las instancias están en Δ. Esto garantiza que las fórmulas cuantificadas universalmente también pertenecen a Δ.

8. Conclusión: Tras descargar todas las premisas, se obtiene $\Delta \vdash S(x) + y = S(x + y)$, y la maximalidad garantiza la pertenencia a Δ.

Significado del lema

`succ_add_Δ` es un resultado técnico de gran importancia para la aritmetización. Establece una propiedad algebraica que es trivial en el modelo estándar pero que debe ser demostrable dentro del sistema formal para que la representación de funciones recursivas funcione correctamente. En particular, esta propiedad es necesaria para manipular expresiones que involucran la función beta de Gödel, donde aparecen sumas y productos de numerales con sucesores. La demostración ilustra el uso del principio de inducción interna del sistema SPA, mostrando que es suficientemente poderoso para derivar propiedades algebraicas no triviales sin apelar a una semántica externa. La verificación de que los axiomas cuantificados universalmente están en Δ utiliza de manera esencial la propiedad de Henkin y la maximalidad, cerrando el círculo entre la saturación existencial y la completitud deductiva.

```
lemma succ_add_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ) (x y : ArithTerm s) :

  ArithForm.eq (ArithTerm.add (ArithTerm.succ x) y) (ArithTerm.succ (ArithTerm.add x y)) ∈ Δ := by

 let v := max_var_term x + 1

 have hv : NotOccur v x := NotOccur_above_max_var_term x v (by omega)


 let Ax0_succ : ArithForm s := add_zero_axiom (ArithTerm.succ x)

 let Ax0_x : ArithForm s := add_zero_axiom x

 let Ax1_succ_body : ArithForm s := add_succ_axiom (ArithTerm.succ x) (ArithTerm.var v)

 let Ax1_succ_forall : ArithForm s := ArithForm.forall_form v Ax1_succ_body

 let Ax1_x_body : ArithForm s := add_succ_axiom x (ArithTerm.var v)

 let Ax1_x_forall : ArithForm s := ArithForm.forall_form v Ax1_x_body


 let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.add (ArithTerm.succ x) (ArithTerm.var v)) (ArithTerm.succ (ArithTerm.add x (ArithTerm.var v)))


 let φ' : ArithForm s := ArithForm.impl Ax0_succ (ArithForm.impl Ax0_x (ArithForm.impl Ax1_succ_forall (ArithForm.impl Ax1_x_forall Eq_v)))


 have h_base : SPA_Deduction_Internal s ∅ (substForm v ArithTerm.zero φ') := by

  rw [substForm_phi_prime x v hv ArithTerm.zero]
```

```
  apply SPA_Deduction_Internal.imp_intro

  apply SPA_Deduction_Internal.imp_intro

  apply SPA_Deduction_Internal.imp_intro

  apply SPA_Deduction_Internal.imp_intro

  let Γ4 : Set (ArithForm s) := insert Ax1_x_forall (insert Ax1_succ_forall (insert Ax0_x (insert Ax0_succ ∅)))

  have h_Ax0_succ : SPA_Deduction_Internal s Γ4 Ax0_succ := SPA_Deduction_Internal.hyp Γ4 Ax0_succ (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

  have h_Ax0_x : SPA_Deduction_Internal s Γ4 Ax0_x := SPA_Deduction_Internal.hyp Γ4 Ax0_x (Or.inr (Or.inr (Or.inl rfl)))

  have h_succ_Ax0_x := SPA_Deduction_Internal.eq_succ Γ4 _ _ h_Ax0_x

  have h_symm_succ := SPA_Deduction_Internal.eq_symm Γ4 _ _ h_succ_Ax0_x

  exact SPA_Deduction_Internal.eq_trans Γ4 _ _ _ h_Ax0_succ h_symm_succ


 have h_step : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v (ArithForm.impl φ' (substForm v (ArithTerm.succ (ArithTerm.var v)) φ'))) := by

  apply SPA_Deduction_Internal.forall_intro

  · rw [substForm_phi_prime x v hv (ArithTerm.succ (ArithTerm.var v))]

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    let Γ5 : Set (ArithForm s) := insert Ax1_x_forall (insert Ax1_succ_forall (insert Ax0_x (insert Ax0_succ (insert φ' ∅))))

    have h_φ' : SPA_Deduction_Internal s Γ5 φ' := SPA_Deduction_Internal.hyp Γ5 φ' (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))

    have h_Ax0_succ : SPA_Deduction_Internal s Γ5 Ax0_succ := SPA_Deduction_Internal.hyp Γ5 Ax0_succ (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

    have h_Ax0_x : SPA_Deduction_Internal s Γ5 Ax0_x := SPA_Deduction_Internal.hyp Γ5 Ax0_x (Or.inr (Or.inr (Or.inl rfl)))
```

have h_Ax1_succ_forall : SPA_Deduction_Internal s Γ5 Ax1_succ_forall := SPA_Deduction_Internal.hyp Γ5 Ax1_succ_forall (Or.inr (Or.inl rfl))

have h_Ax1_x_forall : SPA_Deduction_Internal s Γ5 Ax1_x_forall := SPA_Deduction_Internal.hyp Γ5 Ax1_x_forall (Or.inl rfl)

have h_imp1 := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_φ' h_Ax0_succ

have h_imp2 := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_imp1 h_Ax0_x

have h_imp3 := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_imp2 h_Ax1_succ_forall

have h_Eq_v := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_imp3 h_Ax1_x_forall

have h_succ_Eq_v := SPA_Deduction_Internal.eq_succ Γ5 _ _ h_Eq_v

have h_Ax1_succ_v := SPA_Deduction_Internal.forall_elim Γ5 v Ax1_succ_body (ArithTerm.var v) h_Ax1_succ_forall True.intro

change SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.add (ArithTerm.succ (substTerm v (ArithTerm.var v) x)) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add (ArithTerm.succ (substTerm v (ArithTerm.var v) x)) (substTerm v (ArithTerm.var v) (ArithTerm.var v))))) at h_Ax1_succ_v

rw [substTerm_fresh x v (ArithTerm.var v) hv, substTerm_var_same v (ArithTerm.var v)] at h_Ax1_succ_v

have h_Ax1_x_v := SPA_Deduction_Internal.forall_elim Γ5 v Ax1_x_body (ArithTerm.var v) h_Ax1_x_forall True.intro

change SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.add (substTerm v (ArithTerm.var v) x) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add (substTerm v (ArithTerm.var v) x) (substTerm v (ArithTerm.var v) (ArithTerm.var v))))) at h_Ax1_x_v

rw [substTerm_fresh x v (ArithTerm.var v) hv, substTerm_var_same v (ArithTerm.var v)] at h_Ax1_x_v

have h_succ_Ax1_x_v := SPA_Deduction_Internal.eq_succ Γ5 _ _ h_Ax1_x_v

have h_symm_succ_Ax1_x_v := SPA_Deduction_Internal.eq_symm Γ5 _ _ h_succ_Ax1_x_v

```
    have h_trans1 := SPA_Deduction_Internal.eq_trans Γ5 _ _ _ h_Ax1_succ_v h_succ_Eq_v
    exact SPA_Deduction_Internal.eq_trans Γ5 _ _ _ h_trans1 h_symm_succ_Ax1_x_v
  · intro ψ hψ
    nomatch hψ

 have h_ind : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ') :=
  SPA_Deduction_Internal.induction ∅ φ' v h_base h_step

 have h_ind_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form v φ') :=
  SPA_Deduction_Internal.weaken ∅ _ Δ h_ind (Set.empty_subset _)

 have h_free_phi' : IsFreeFor v y φ' := by
  dsimp [φ', Ax0_succ, Ax0_x, Ax1_succ_forall, Ax1_x_forall, Eq_v, add_zero_axiom, add_succ_axiom, IsFreeFor]
  have h_vv1 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Ax1_succ_body) = True := if_pos rfl
  have h_vv2 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Ax1_x_body) = True := if_pos rfl
  rw [h_vv1, h_vv2]
  refine ⟨trivial, ⟨trivial, ⟨trivial, ⟨trivial, trivial⟩⟩⟩⟩
 have h_elim := SPA_Deduction_Internal.forall_elim Δ v φ' y h_ind_Δ h_free_phi'

 rw [substForm_phi_prime x v hv y] at h_elim

 have h_Ax0_succ_Δ : SPA_Deduction_Internal s Δ Ax0_succ := SPA_Deduction_Internal.hyp Δ _ (h_add_zero (ArithTerm.succ x))
 have h_Ax0_x_Δ : SPA_Deduction_Internal s Δ Ax0_x := SPA_Deduction_Internal.hyp Δ _ (h_add_zero x)
```

```
  have h_elim_1 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim h_Ax0_succ_Δ
  have h_elim_2 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim_1 h_Ax0_x_Δ

  have h_Ax1_succ_forall_in : Ax1_succ_forall ∈ Δ := by
    by_contra h_not_in
    have h_neg_in : ArithForm.neg Ax1_succ_forall ∈ Δ := by
      cases h_max Ax1_succ_forall with
      | inl h => contradiction
      | inr h => exact h
    have h_neg_double : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg (add_succ_axiom (ArithTerm.succ x) (ArithTerm.var v))))) ∈ Δ := by
      rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
    obtain ⟨t, h_witness⟩ := h_henkin v (ArithForm.neg (add_succ_axiom (ArithTerm.succ x) (ArithTerm.var v))) h_neg_double
    change ArithForm.neg (ArithForm.eq (ArithTerm.add (ArithTerm.succ (substTerm v t.val x)) (ArithTerm.succ (substTerm v t.val (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add (ArithTerm.succ (substTerm v t.val x)) (substTerm v t.val (ArithTerm.var v))))) ∈ Δ at h_witness
    rw [substTerm_fresh x v t.val hv, substTerm_var_same v t.val] at h_witness
    have h_pos := h_add_succ (ArithTerm.succ x) t.val
    have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
      SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ h_witness)
    exact h_con h_bot

  have h_Ax1_x_forall_in : Ax1_x_forall ∈ Δ := by
    by_contra h_not_in
    have h_neg_in : ArithForm.neg Ax1_x_forall ∈ Δ := by
      cases h_max Ax1_x_forall with
```

```
    | inl h => contradiction
    | inr h => exact h
  have h_neg_double : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg (add_succ_axiom x (ArithTerm.var v))))) ∈ Δ := by
    rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
  obtain ⟨t, h_witness⟩ := h_henkin v (ArithForm.neg (add_succ_axiom x (ArithTerm.var v))) h_neg_double
  change ArithForm.neg (ArithForm.eq (ArithTerm.add (substTerm v t.val x) (ArithTerm.succ (substTerm v t.val (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add (substTerm v t.val x) (substTerm v t.val (ArithTerm.var v))))) ∈ Δ at h_witness
  rw [substTerm_fresh x v t.val hv, substTerm_var_same v t.val] at h_witness
  have h_pos := h_add_succ x t.val
  have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
    SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ h_witness)
  exact h_con h_bot

 have h_Ax1_succ_forall_Δ : SPA_Deduction_Internal s Δ Ax1_succ_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_succ_forall_in
 have h_Ax1_x_forall_Δ : SPA_Deduction_Internal s Δ Ax1_x_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_x_forall_in

 have h_elim_3 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim_2 h_Ax1_succ_forall_Δ
 have h_final_ded := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim_3 h_Ax1_x_forall_Δ

 exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final_ded
```

Lema `zero_add_Δ`: Elemento neutro derecho de la adición en Δ

Se demuestra que 0 es neutro a izquierda para la adición dentro de Δ: para cualquier término y, la suma $0 + y$ es demostrablemente igual a y:

$$ \forall y \in \text{ArithTerm } s,\ \Delta \vdash 0 + y = y $$

Estructura de la demostración

La prueba sigue el mismo patrón de inducción interna utilizado en `succ_add_Δ`, pero aplicado a una fórmula más simple.

1. Variable fresca y construcción de la fórmula de inducción: Se elige $v := \max_var_term(y) + 1$ (fresca respecto a y) y se define:

$$ \varphi'(v) := \text{Ax0}_0 \to (\forall v\ \text{Ax1}_0(v) \to (0 + v = v)) $$

donde:

- $\text{Ax0}_0 = (0 + 0 = 0)$ (axioma add_zero para 0)

- $\text{Ax1}_0(v) = \forall v\ (0 + S(v) = S(0 + v))$ (axioma add_succ para 0)

2. Caso base ($v := 0$): $\varphi'(0)$ se reduce, tras sustitución, a probar $0 + 0 = 0$ bajo los axiomas, lo cual es exactamente Ax0_0.

3. Paso inductivo ($v \to S(v)$): Asumiendo $\varphi'(v)$ (que proporciona $0 + v = v$) y los axiomas, se debe probar $0 + S(v) = S(v)$. La cadena de igualdades es:

- Por Ax1_0: $0 + S(v) = S(0 + v)$

- Por la hipótesis inductiva $\varphi'(v)$: $0 + v = v$, de donde $S(0 + v) = S(v)$ por congruencia

- Por transitividad: $0 + S(v) = S(v)$.

4. Aplicación de la inducción y descarga de axiomas: El esquema de inducción proporciona $\forall v\ \varphi'(v)$. Instanciando con y (verificando que y es libre para v en φ') se obtiene $\varphi'(y) \in \Delta$. Las premisas de $\varphi'(y)$ son Ax0_0 y $\forall v\ \text{Ax1}_0(v)$, ambas pertenecientes a Δ por las hipótesis (la segunda requiere el mismo argumento de maximalidad y Henkin que en `succ_add_Δ`). Por eliminación de la implicación se concluye $0 + y = y \in \Delta$.

Significado

`zero_add_Δ` completa las propiedades básicas de la adición en Δ, estableciendo que 0 es efectivamente el elemento neutro tanto a izquierda como a derecha (el lado derecho ya estaba dado por `add_zero_axiom`). Esta propiedad, aunque elemental en el modelo estándar, debe ser demostrable dentro del sistema formal para que la aritmetización funcione correctamente. En particular, es necesaria para simplificar expresiones que involucran sumas con cero en la representación de funciones recursivas.

La estructura de la demostración, análoga a la de `succ_add_Δ`, confirma que el principio de inducción interna del sistema SPA es una herramienta sistemática para derivar propiedades algebraicas de los términos. La repetición del patrón —variable fresca, fórmula de inducción con implicaciones, caso base y paso inductivo, aplicación del esquema `induction`, y descarga de axiomas mediante maximalidad y Henkin— sugiere que este método es generalizable a otras identidades aritméticas,

constituyendo una maquinaria robusta para el desarrollo de la aritmética formal dentro de Δ.

```
--
================================================================

-- LEMA DE DERECHA DE CERO PARA LA ADICIÓN: 0 + y = y EN Δ

--
================================================================


lemma zero_add_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

   (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

   (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ) (y : ArithTerm s) :

   ArithForm.eq (ArithTerm.add ArithTerm.zero y) y ∈ Δ := by

 let v := max_var_term y + 1

 have hv : NotOccur v y := NotOccur_above_max_var_term y v (by omega)


 let Ax0_zero : ArithForm s := add_zero_axiom (ArithTerm.zero : ArithTerm s)

 let Ax1_zero_body : ArithForm s := add_succ_axiom (ArithTerm.zero : ArithTerm s) (ArithTerm.var v)

 let Ax1_zero_forall : ArithForm s := ArithForm.forall_form v Ax1_zero_body


 let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.add ArithTerm.zero (ArithTerm.var v)) (ArithTerm.var v)


 let φ' : ArithForm s := ArithForm.impl Ax0_zero (ArithForm.impl Ax1_zero_forall Eq_v)
```

have h_subst_phi' : ∀ u : ArithTerm s, substForm v u φ' =

ArithForm.impl Ax0_zero (ArithForm.impl Ax1_zero_forall (ArithForm.eq (ArithTerm.add ArithTerm.zero u) u)) := by

intro u

dsimp [φ', Ax0_zero, Ax1_zero_forall, Ax1_zero_body, Eq_v, add_zero_axiom, add_succ_axiom, substForm, substTerm]

have h_if : (if v = v then u else ArithTerm.var v) = u := if_pos rfl

rw [h_if]

have h_if_forall : (if v = v then

ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.zero : ArithTerm s) (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add ArithTerm.zero (ArithTerm.var v))))

else

ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.zero : ArithTerm s) (ArithTerm.succ u)) (ArithTerm.succ (ArithTerm.add ArithTerm.zero u)))) =

ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.zero : ArithTerm s) (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add ArithTerm.zero (ArithTerm.var v)))) := if_pos rfl

rw [h_if_forall]

have h_base : SPA_Deduction_Internal s ∅ (substForm v ArithTerm.zero φ') := by

rw [h_subst_phi' ArithTerm.zero]

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

let Γ2 : Set (ArithForm s) := insert Ax1_zero_forall (insert Ax0_zero ∅)

exact SPA_Deduction_Internal.hyp Γ2 Ax0_zero (Or.inr (Or.inl rfl))

have h_step : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v (ArithForm.impl φ' (substForm v (ArithTerm.succ (ArithTerm.var v)) φ'))) := by

apply SPA_Deduction_Internal.forall_intro

· rw [h_subst_phi' (ArithTerm.succ (ArithTerm.var v))]

```
    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    let Γ3 : Set (ArithForm s) := insert Ax1_zero_forall (insert Ax0_zero (insert φ' ∅))

    have h_φ' : SPA_Deduction_Internal s Γ3 φ' := SPA_Deduction_Internal.hyp Γ3 φ' (Or.inr (Or.inr (Or.inl rfl)))

    have h_Ax0_zero : SPA_Deduction_Internal s Γ3 Ax0_zero := SPA_Deduction_Internal.hyp Γ3 Ax0_zero (Or.inr (Or.inl rfl))

    have h_Ax1_zero_forall : SPA_Deduction_Internal s Γ3 Ax1_zero_forall := SPA_Deduction_Internal.hyp Γ3 Ax1_zero_forall (Or.inl rfl)


    have h_imp1 := SPA_Deduction_Internal.imp_elim Γ3 _ _ h_φ' h_Ax0_zero

    have h_Eq_v := SPA_Deduction_Internal.imp_elim Γ3 _ _ h_imp1 h_Ax1_zero_forall


    have h_succ_Eq_v := SPA_Deduction_Internal.eq_succ Γ3 _ _ h_Eq_v


    have h_Ax1_v := SPA_Deduction_Internal.forall_elim Γ3 v Ax1_zero_body (ArithTerm.var v) h_Ax1_zero_forall True.intro

    change SPA_Deduction_Internal s Γ3 (ArithForm.eq (ArithTerm.add ArithTerm.zero (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add ArithTerm.zero (substTerm v (ArithTerm.var v) (ArithTerm.var v))))) at h_Ax1_v

    rw [substTerm_var_same v (ArithTerm.var v)] at h_Ax1_v


    exact SPA_Deduction_Internal.eq_trans Γ3 _ _ _ h_Ax1_v h_succ_Eq_v

  · intro ψ hψ

    nomatch hψ


 have h_ind : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ') :=

  SPA_Deduction_Internal.induction ∅ φ' v h_base h_step
```

```
  have h_ind_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form v φ') :=
    SPA_Deduction_Internal.weaken ∅ _ Δ h_ind (Set.empty_subset _)

  have h_free_phi' : IsFreeFor v y φ' := by
    dsimp [φ', Ax0_zero, Ax1_zero_forall, Eq_v, add_zero_axiom, add_succ_axiom, IsFreeFor]
    have h_if_free : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Ax1_zero_body) = True := if_pos rfl
    rw [h_if_free]
    exact ⟨True.intro, True.intro, True.intro⟩

  have h_elim := SPA_Deduction_Internal.forall_elim Δ v φ' y h_ind_Δ h_free_phi'
  rw [h_subst_phi' y] at h_elim

  have h_Ax0_zero_Δ : SPA_Deduction_Internal s Δ Ax0_zero := SPA_Deduction_Internal.hyp Δ _ (h_add_zero ArithTerm.zero)
  have h_elim_1 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim h_Ax0_zero_Δ

  have h_Ax1_zero_forall_in : Ax1_zero_forall ∈ Δ := by
    by_contra h_not_in
    have h_neg_in : ArithForm.neg Ax1_zero_forall ∈ Δ := by
      cases h_max Ax1_zero_forall with
      | inl h => contradiction
      | inr h => exact h
    have h_neg_double : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg (add_succ_axiom ArithTerm.zero (ArithTerm.var v))))) ∈ Δ := by
      rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
    obtain ⟨t, h_witness⟩ := h_henkin v (ArithForm.neg (add_succ_axiom ArithTerm.zero (ArithTerm.var v))) h_neg_double
```

```
  change ArithForm.neg (ArithForm.eq (ArithTerm.add ArithTerm.zero (ArithTerm.succ (substTerm v t.val (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add ArithTerm.zero (substTerm v t.val (ArithTerm.var v))))) ∈ Δ at h_witness

  rw [substTerm_var_same v t.val] at h_witness

  have h_pos := h_add_succ ArithTerm.zero t.val

  have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=

   SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ h_witness)

  exact h_con h_bot


 have h_Ax1_zero_forall_Δ : SPA_Deduction_Internal s Δ Ax1_zero_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_zero_forall_in

 have h_final_ded := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim_1 h_Ax1_zero_forall_Δ


 exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final_ded
```

Lema `add_comm_Δ`: Conmutatividad general de la adición en Δ

Se demuestra la propiedad conmutativa de la adición para términos arbitrarios dentro de Δ: para cualesquiera términos x e y, la suma es demostrablemente conmutativa:

$$ \forall x, y \in \text{ArithTerm } s,\ \Delta \vdash x + y = y + x $$

Estructura de la demostración

La prueba sigue el patrón de inducción interna ya establecido en los lemas anteriores, pero con una fórmula de inducción más compleja que incorpora múltiples hipótesis auxiliares necesarias para completar el paso inductivo.

1. Variable fresca y construcción de φ': Se elige $v := \max_var_term(x) + \max_var_term(y) + 1$, fresca respecto a ambos términos. La fórmula de inducción es:

$$ \varphi'(v) := \text{Ax0}_x \to (0 + x = x) \to (\forall v\ \text{Ax1}_x(v)) \to (\forall v\ \text{SuccAdd}(v)) \to (x + v = v + x) $$

donde:

- $\text{Ax0}_x = (x + 0 = x)$

- $(0 + x = x)$ es la propiedad del neutro izquierdo (lema `zero_add_Δ`)

- $\forall v\ \text{Ax1}_x(v) = \forall v\ (x + S(v) = S(x + v))$

- $\forall v\ \text{SuccAdd}(v) = \forall v\ (S(v) + x = S(v + x))$ (lema `succ_add_Δ`)

- La conclusión es $x + v = v + x$.

2. Caso base ($v := 0$): Tras sustitución, se debe probar $x + 0 = 0 + x$ bajo las hipótesis. Por Ax0_x: $x + 0 = x$. Por `zero_add_Δ` y simetría: $x = 0 + x$. Por transitividad: $x + 0 = 0 + x$.

3. Paso inductivo ($v \to S(v)$): Asumiendo $\varphi'(v)$ (que proporciona $x + v = v + x$) y todas las hipótesis, se debe probar $x + S(v) = S(v) + x$. La cadena de igualdades es:

- Por $\text{Ax1}_x(v)$: $x + S(v) = S(x + v)$

- Por hipótesis inductiva y congruencia: $S(x + v) = S(v + x)$

- Por $\text{SuccAdd}(v)$ y simetría: $S(v + x) = S(v) + x$

- Por transitividad: $x + S(v) = S(v) + x$.

4. Instanciación y descarga de axiomas: La inducción proporciona $\forall v\ \varphi'(v)$. Instanciando con y (verificando libertad de sustitución) se obtiene $\varphi'(y) \in \Delta$. Las cuatro premisas de $\varphi'(y)$ se descargan una a una:

- Ax0_x está en Δ por `h_add_zero`.

- $(0 + x = x)$ está en Δ por `zero_add_Δ`.

- $\forall v\ \text{Ax1}_x(v)$ y $\forall v\ \text{SuccAdd}(v)$ se demuestra que pertenecen a Δ usando maximalidad y Henkin: si no pertenecieran, su negación estaría en Δ, y por Henkin existiría un contraejemplo que contradice `h_add_succ` o `succ_add_Δ` respectivamente.

- Tras cuatro eliminaciones de la implicación, se obtiene $x + y = y + x \in \Delta$.

Significado y arquitectura de la prueba

`add_comm_Δ` es un resultado fundamental que completa las propiedades algebraicas básicas de la adición en Δ. Junto con `add_assoc_numeral_Δ`, `zero_add_Δ` y `succ_add_Δ`, establece que la adición en el lenguaje objeto se comporta como una operación conmutativa y asociativa con elemento neutro, al menos para las igualdades demostrables en Δ.

La estructura de la demostración pone de manifiesto una arquitectura en capas:

1. **Nivel base:** Los axiomas `add_zero_axiom` y `add_succ_axiom` proporcionan las propiedades elementales de la adición.

2. **Nivel intermedio:** Lemas como `zero_add_Δ` y `succ_add_Δ` derivan propiedades adicionales usando inducción interna.

3. **Nivel superior:** `add_comm_Δ` combina las propiedades de los niveles anteriores, también mediante inducción interna, para obtener la conmutatividad plena.

Esta dependencia jerárquica refleja el desarrollo estándar de la aritmética a partir de los axiomas de Peano, pero todo el razonamiento se lleva a cabo dentro del sistema formal SPA, sin apelar a una semántica externa. La verificación de que las fórmulas cuantificadas universalmente pertenecen a Δ (usando maximalidad y Henkin) es un patrón recurrente que será reutilizado en la demostración de representabilidad de funciones más complejas, donde se necesita garantizar que ciertas propiedades valen para todos los numerales.

```
--
============================================================

-- CONMUTATIVIDAD GENERAL DE LA ADICIÓN EN Δ: x + y = y + x

--
============================================================



lemma add_comm_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con :
Consistent Δ) (h_henkin : HenkinSet Δ)

   (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

   (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ) (x y : ArithTerm s) :

   ArithForm.eq (ArithTerm.add x y) (ArithTerm.add y x) ∈ Δ := by

  let v := max_var_term x + max_var_term y + 1

  have hv_x : NotOccur v x := NotOccur_above_max_var_term x v (by omega)

  have hv_y : NotOccur v y := NotOccur_above_max_var_term y v (by omega)


  let Ax0_x : ArithForm s := add_zero_axiom x

  let ZeroAdd_x : ArithForm s := ArithForm.eq (ArithTerm.add ArithTerm.zero x) x
```

```
  let Ax1_x_body : ArithForm s := add_succ_axiom x (ArithTerm.var v)

  let Ax1_x_forall : ArithForm s := ArithForm.forall_form v Ax1_x_body


  let SuccAdd_body : ArithForm s := ArithForm.eq (ArithTerm.add (ArithTerm.succ
(ArithTerm.var v)) x) (ArithTerm.succ (ArithTerm.add (ArithTerm.var v) x))

  let SuccAdd_forall : ArithForm s := ArithForm.forall_form v SuccAdd_body


  let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.add x (ArithTerm.var v))
(ArithTerm.add (ArithTerm.var v) x)


  let φ' : ArithForm s := ArithForm.impl Ax0_x (ArithForm.impl ZeroAdd_x
(ArithForm.impl Ax1_x_forall (ArithForm.impl SuccAdd_forall Eq_v)))


  have h_subst_phi' : ∀ u : ArithTerm s, substForm v u φ' =

    ArithForm.impl Ax0_x (ArithForm.impl ZeroAdd_x (ArithForm.impl Ax1_x_forall
(ArithForm.impl SuccAdd_forall (ArithForm.eq (ArithTerm.add x u) (ArithTerm.add u
x))))) := by

    intro u

    dsimp [φ', Ax0_x, ZeroAdd_x, Ax1_x_forall, SuccAdd_forall, Ax1_x_body,
SuccAdd_body, Eq_v, add_zero_axiom, add_succ_axiom, substForm, substTerm]

    have h_fresh_x := substTerm_fresh x v u hv_x

    rw [h_fresh_x]

    have h_if_u : (if v = v then u else ArithTerm.var v) = u := if_pos rfl

    rw [h_if_u]

    have h_if_forall1 : (if v = v then

      ArithForm.forall_form v (ArithForm.eq (ArithTerm.add x (ArithTerm.succ
(ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add x (ArithTerm.var v))))

     else

      ArithForm.forall_form v (ArithForm.eq (ArithTerm.add x (ArithTerm.succ u))
(ArithTerm.succ (ArithTerm.add x u)))) =
```

ArithForm.forall_form v (ArithForm.eq (ArithTerm.add x (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add x (ArithTerm.var v)))) := if_pos rfl

rw [h_if_forall1]

have h_if_forall2 : (if v = v then

ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.succ (ArithTerm.var v)) x) (ArithTerm.succ (ArithTerm.add (ArithTerm.var v) x)))

else

ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.succ u) x) (ArithTerm.succ (ArithTerm.add u x)))) =

ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.succ (ArithTerm.var v)) x) (ArithTerm.succ (ArithTerm.add (ArithTerm.var v) x))) := if_pos rfl

rw [h_if_forall2]

have h_base : SPA_Deduction_Internal s ∅ (substForm v ArithTerm.zero φ') := by

rw [h_subst_phi' ArithTerm.zero]

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

let Γ4 : Set (ArithForm s) := insert SuccAdd_forall (insert Ax1_x_forall (insert ZeroAdd_x (insert Ax0_x ∅)))

have h_Ax0_x : SPA_Deduction_Internal s Γ4 Ax0_x := SPA_Deduction_Internal.hyp Γ4 Ax0_x (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

have h_ZeroAdd_x : SPA_Deduction_Internal s Γ4 ZeroAdd_x := SPA_Deduction_Internal.hyp Γ4 ZeroAdd_x (Or.inr (Or.inr (Or.inl rfl)))

have h_symm_ZeroAdd := SPA_Deduction_Internal.eq_symm Γ4 _ _ h_ZeroAdd_x

exact SPA_Deduction_Internal.eq_trans Γ4 _ _ _ h_Ax0_x h_symm_ZeroAdd

have h_step : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v (ArithForm.impl φ' (substForm v (ArithTerm.succ (ArithTerm.var v)) φ'))) := by

```
  apply SPA_Deduction_Internal.forall_intro
  · rw [h_subst_phi' (ArithTerm.succ (ArithTerm.var v))]
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    let Γ5 : Set (ArithForm s) := insert SuccAdd_forall (insert Ax1_x_forall (insert ZeroAdd_x (insert Ax0_x (insert φ' ∅))))
    have h_φ' : SPA_Deduction_Internal s Γ5 φ' := SPA_Deduction_Internal.hyp Γ5 φ' (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))
    have h_Ax0_x : SPA_Deduction_Internal s Γ5 Ax0_x := SPA_Deduction_Internal.hyp Γ5 Ax0_x (Or.inr (Or.inr (Or.inr (Or.inl rfl))))
    have h_ZeroAdd_x : SPA_Deduction_Internal s Γ5 ZeroAdd_x := SPA_Deduction_Internal.hyp Γ5 ZeroAdd_x (Or.inr (Or.inr (Or.inl rfl)))
    have h_Ax1_x_forall : SPA_Deduction_Internal s Γ5 Ax1_x_forall := SPA_Deduction_Internal.hyp Γ5 Ax1_x_forall (Or.inr (Or.inl rfl))
    have h_SuccAdd_forall : SPA_Deduction_Internal s Γ5 SuccAdd_forall := SPA_Deduction_Internal.hyp Γ5 SuccAdd_forall (Or.inl rfl)

    have h_imp1 := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_φ' h_Ax0_x
    have h_imp2 := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_imp1 h_ZeroAdd_x
    have h_imp3 := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_imp2 h_Ax1_x_forall
    have h_Eq_v := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_imp3 h_SuccAdd_forall

    have h_succ_Eq_v := SPA_Deduction_Internal.eq_succ Γ5 _ _ h_Eq_v

    have h_Ax1_v := SPA_Deduction_Internal.forall_elim Γ5 v Ax1_x_body (ArithTerm.var v) h_Ax1_x_forall True.intro
    change SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.add (substTerm v (ArithTerm.var v) x) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var
```

v)))) (ArithTerm.succ (ArithTerm.add (substTerm v (ArithTerm.var v) x) (substTerm v (ArithTerm.var v) (ArithTerm.var v))))) at h_Ax1_v

rw [substTerm_fresh x v (ArithTerm.var v) hv_x, substTerm_var_same v (ArithTerm.var v)] at h_Ax1_v

have h_SuccAdd_v := SPA_Deduction_Internal.forall_elim Γ5 v SuccAdd_body (ArithTerm.var v) h_SuccAdd_forall True.intro

change SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.add (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) x)) (ArithTerm.succ (ArithTerm.add (substTerm v (ArithTerm.var v) (ArithTerm.var v)) (substTerm v (ArithTerm.var v) x)))) at h_SuccAdd_v

rw [substTerm_fresh x v (ArithTerm.var v) hv_x, substTerm_var_same v (ArithTerm.var v)] at h_SuccAdd_v

have h_symm_SuccAdd_v := SPA_Deduction_Internal.eq_symm Γ5 _ _ h_SuccAdd_v

have h_trans1 := SPA_Deduction_Internal.eq_trans Γ5 _ _ _ h_Ax1_v h_succ_Eq_v

exact SPA_Deduction_Internal.eq_trans Γ5 _ _ _ h_trans1 h_symm_SuccAdd_v

· intro ψ hψ

nomatch hψ

have h_ind : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ') :=

SPA_Deduction_Internal.induction ∅ φ' v h_base h_step

have h_ind_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form v φ') :=

SPA_Deduction_Internal.weaken ∅ _ Δ h_ind (Set.empty_subset _)

have h_free_phi' : IsFreeFor v y φ' := by

dsimp [φ', Ax0_x, ZeroAdd_x, Ax1_x_forall, SuccAdd_forall, Eq_v, add_zero_axiom, add_succ_axiom, IsFreeFor]

have h_if1 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Ax1_x_body) = True := if_pos rfl

```
  have h_if2 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y SuccAdd_body) = True := if_pos rfl
  rw [h_if1, h_if2]
  exact ⟨True.intro, True.intro, True.intro, True.intro, True.intro⟩

 have h_elim := SPA_Deduction_Internal.forall_elim Δ v φ' y h_ind_Δ h_free_phi'
 rw [h_subst_phi' y] at h_elim

 have h_Ax0_x_Δ : SPA_Deduction_Internal s Δ Ax0_x := SPA_Deduction_Internal.hyp Δ _ (h_add_zero x)
 have h_ZeroAdd_x_in := zero_add_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ x
 have h_ZeroAdd_x_Δ : SPA_Deduction_Internal s Δ ZeroAdd_x := SPA_Deduction_Internal.hyp Δ _ h_ZeroAdd_x_in

 have h_elim_1 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim h_Ax0_x_Δ
 have h_elim_2 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim_1 h_ZeroAdd_x_Δ

 have h_Ax1_x_forall_in : Ax1_x_forall ∈ Δ := by
  by_contra h_not_in
  have h_neg_in : ArithForm.neg Ax1_x_forall ∈ Δ := by
   cases h_max Ax1_x_forall with
   | inl h => contradiction
   | inr h => exact h
  have h_neg_double : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg (add_succ_axiom x (ArithTerm.var v))))) ∈ Δ := by
   rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
  obtain ⟨t, h_witness⟩ := h_henkin v (ArithForm.neg (add_succ_axiom x (ArithTerm.var v))) h_neg_double
  change ArithForm.neg (ArithForm.eq (ArithTerm.add (substTerm v t.val x) (ArithTerm.succ (substTerm v t.val (ArithTerm.var v)))) (ArithTerm.succ
```

(ArithTerm.add (substTerm v t.val x) (substTerm v t.val (ArithTerm.var v))))) ∈ Δ at h_witness

rw [substTerm_fresh x v t.val hv_x, substTerm_var_same v t.val] at h_witness

have h_pos := h_add_succ x t.val

have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=

SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ h_witness)

exact h_con h_bot

have h_SuccAdd_forall_in : SuccAdd_forall ∈ Δ := by

by_contra h_not_in

have h_neg_in : ArithForm.neg SuccAdd_forall ∈ Δ := by

cases h_max SuccAdd_forall with

| inl h => contradiction

| inr h => exact h

have h_neg_double : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg SuccAdd_body))) ∈ Δ := by

rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]

obtain ⟨t, h_witness⟩ := h_henkin v (ArithForm.neg SuccAdd_body) h_neg_double

change ArithForm.neg (ArithForm.eq (ArithTerm.add (ArithTerm.succ (substTerm v t.val (ArithTerm.var v))) (substTerm v t.val x)) (ArithTerm.succ (ArithTerm.add (substTerm v t.val (ArithTerm.var v)) (substTerm v t.val x)))) ∈ Δ at h_witness

rw [substTerm_fresh x v t.val hv_x, substTerm_var_same v t.val] at h_witness

have h_pos := succ_add_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ t.val x

have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=

SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ h_witness)

exact h_con h_bot

have h_Ax1_x_forall_Δ : SPA_Deduction_Internal s Δ Ax1_x_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_x_forall_in

have h_SuccAdd_forall_Δ : SPA_Deduction_Internal s Δ SuccAdd_forall := SPA_Deduction_Internal.hyp Δ _ h_SuccAdd_forall_in

have h_elim_3 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim_2 h_Ax1_x_forall_Δ

have h_final_ded := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim_3 h_SuccAdd_forall_Δ

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final_ded

Lema `M_term_eq_natToTerm`: Igualdad del término módulo con su representación numeral

Se demuestra una igualdad fundamental para la representabilidad de la función beta de Gödel: el término compuesto $M_{\text{term}} = S(S(\overline{I}) \cdot \overline{B})$ es demostrablemente igual al numeral $\overline{M}$ donde $M = 1 + (I + 1) \cdot B$:

$$ \forall I, B \in \mathbb{N},\ \Delta \vdash \overline{1 + (I + 1) \cdot B} = S(S(\overline{I}) \cdot \overline{B}) $$

Estructura de la demostración

La prueba construye una cadena de igualdades en Δ que conecta el numeral $\overline{M}$ con el término compuesto M_{term}:

Paso 1: Descomposición de M. Se observa que $M = (I + 1) \cdot B + 1$, por lo que $\overline{M} = S(\overline{(I + 1) \cdot B})$ por definición de `natToTerm`. Esta igualdad es puramente definicional y se establece por reflexividad.

Paso 2: Homomorfismo de la multiplicación. El lema `natToTerm_mul_hom` proporciona $\overline{(I + 1) \cdot B} = \overline{I + 1} \cdot \overline{B} \in \Delta$.

Paso 3: Descomposición de $\overline{I + 1}$. Por definición, $\overline{I + 1} = S(\overline{I})$. Esta igualdad es reflexiva en Δ.

Paso 4: Congruencia de la multiplicación. Combinando el paso 3 con la reflexividad de $\overline{B}$, se aplica `eq_mul_Δ` para obtener $\overline{I + 1} \cdot \overline{B} = S(\overline{I}) \cdot \overline{B} \in \Delta$.

Paso 5: Transitividad. De los pasos 2 y 4, por transitividad: $\overline{(I + 1) \cdot B} = S(\overline{I}) \cdot \overline{B} \in \Delta$.

Paso 6: Congruencia del sucesor. Envolviendo el paso 5 con el constructor `succ`: $S(\overline{(I + 1) \cdot B}) = S(S(\overline{I}) \cdot \overline{B}) \in \Delta$.

Paso 7: Conexión con $\overline{M}$. Del paso 1, $\overline{M} = S(\overline{(I + 1) \cdot B}) \in \Delta$ (reflexividad).

Paso 8: Transitividad final. Combinando los pasos 6 y 7: $\overline{M} = S(S(\overline{I}) \cdot \overline{B}) = M_{\text{term}} \in \Delta$.

Significado para la función beta de Gödel

Este lema es una pieza esencial en la representabilidad de la función beta. La función beta se define mediante el residuo de la división:

$$ \beta(a, b, i) = a \bmod (1 + (i + 1) \cdot b) $$

En la fórmula `BetaForm`, el módulo $M = 1 + (i + 1) \cdot b$ aparece representado de dos maneras distintas en diferentes partes de la fórmula: como el numeral $\overline{M}$ (en la igualdad que define el cociente q) y como el término compuesto $S(S(\overline{i}) \cdot \overline{b})$ (en la condición de orden $r < M_{\text{term}}$). El lema `M_term_eq_natToTerm` establece que ambas representaciones son demostrablemente iguales en Δ, lo cual es indispensable para conectar las distintas partes de la fórmula y demostrar que `BetaForm` captura correctamente la relación de la función beta. Sin esta igualdad, no sería posible deducir que el residuo r satisface simultáneamente $a = q \cdot M + r$ y $r < M$ internamente.

--
==

-- TEOREMA DE REPRESENTABILIDAD DE LA FUNCIÓN BETA DE GÖDEL

--
==

--
==

-- LEMAS AUXILIARES PARA beta_representable

```
--
================================================================


/-- M_term = natToTerm M en Δ, donde M = 1 + (I+1)*B -/

lemma M_term_eq_natToTerm {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)

   (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

   (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

   (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

   (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

   (I B : Nat) :

   let M := 1 + (I + 1) * B

   let M_term := ArithTerm.succ (ArithTerm.mul (ArithTerm.succ (natToTerm (s := s)
I)) (natToTerm (s := s) B))

   ArithForm.eq (natToTerm (s := s) M) M_term ∈ Δ := by

  intro M M_term

  -- Paso 1: natToTerm M = succ (natToTerm ((I+1)*B)) por definición (M = (I+1)*B +
1)

  have h_M_def : M = (I + 1) * B + 1 := by omega

  have h_natToTerm_M : natToTerm (s := s) M = ArithTerm.succ (natToTerm (s := s) ((I
+ 1) * B)) := by

   rw [h_M_def]

   rfl

  -- Paso 2: natToTerm ((I+1)*B) = mul (natToTerm (I+1)) (natToTerm B) en Δ

  have h_mul_hom := natToTerm_mul_hom Δ h_max h_con h_add_zero h_add_succ
h_mul_zero h_mul_succ (I + 1) B

  -- Paso 3: natToTerm (I+1) = succ (natToTerm I) (definicional)

  have h_I1 : natToTerm (s := s) (I + 1) = ArithTerm.succ (natToTerm (s := s) I) := rfl

  -- Paso 4: eq (natToTerm (I+1)) (succ (natToTerm I)) en Δ

  have h_I1_eq : ArithForm.eq (natToTerm (s := s) (I + 1)) (ArithTerm.succ (natToTerm
(s := s) I)) ∈ Δ := by
```

```
  rw [h_I1]
  exact eq_refl_in_Δ Δ h_max h_con _
 -- Paso 5: Por congruencia de mul:
 -- mul (natToTerm (I+1)) (natToTerm B) = mul (succ (natToTerm I)) (natToTerm B) en Δ
 have h_refl_B := eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) B)
 have h_mul_congr := eq_mul_Δ h_max h_con h_I1_eq h_refl_B
 -- Paso 6: natToTerm ((I+1)*B) = mul (succ (natToTerm I)) (natToTerm B) en Δ
 have h_mul_trans := eq_trans_Δ h_max h_con h_mul_hom h_mul_congr
 -- Paso 7: succ (natToTerm ((I+1)*B)) = succ (mul (succ (natToTerm I)) (natToTerm B)) en Δ
 have h_succ_congr := eq_succ_Δ h_max h_con h_mul_trans
 -- Paso 8: natToTerm M = succ (natToTerm ((I+1)*B)) en Δ (por h_natToTerm_M)
 have h_natToTerm_M_eq : ArithForm.eq (natToTerm (s := s) M) (ArithTerm.succ (natToTerm (s := s) ((I + 1) * B))) ∈ Δ := by
  rw [h_natToTerm_M]
  exact eq_refl_in_Δ Δ h_max h_con _
 -- Paso 9: Transitividad final: natToTerm M = M_term en Δ
 exact eq_trans_Δ h_max h_con h_natToTerm_M_eq h_succ_congr
```

Lemas auxiliares para la representabilidad de la función beta

Se demuestran dos lemas técnicos que conectan las diferentes representaciones de la división y el orden en la fórmula `BetaForm`. Estos lemas son esenciales para

establecer que la fórmula captura correctamente la relación $A = Q \cdot M + R$ con $R < M$, donde $M = 1 + (I+1) \cdot B$.

Lema `eq_A_QMR_term`: Igualdad de la división con el término módulo compuesto

Establece que si externamente se satisface la igualdad $A = Q \cdot M + R$ (donde $M = 1 + (I+1) \cdot B$), entonces internamente Δ demuestra la igualdad correspondiente utilizando el término compuesto M_{term}:

$$ \forall A, B, I, R, Q \in \mathbb{N},\ A = Q \cdot (1 + (I+1) \cdot B) + R \implies $$

$$ \Delta \vdash \overline{A} = \overline{Q} \cdot M_{\text{term}} + \overline{R} $$

donde $M_{\text{term}} = S(S(\overline{I}) \cdot \overline{B})$.

Estructura de la demostración

La prueba construye una cadena de igualdades en Δ que transforma $\overline{A}$ en la expresión $\overline{Q} \cdot M_{\text{term}} + \overline{R}$:

Paso 1: Reescritura de A. Dado que $A = Q \cdot M + R$, se tiene $\overline{A} = \overline{Q \cdot M + R} \in \Delta$ por reflexividad.

Paso 2: Homomorfismo de la adición. $\overline{Q \cdot M + R} = \overline{Q \cdot M} + \overline{R} \in \Delta$ por `natToTerm_add_hom`.

Paso 3: Homomorfismo de la multiplicación. $\overline{Q \cdot M} = \overline{Q} \cdot \overline{M} \in \Delta$ por `natToTerm_mul_hom`.

Paso 4: Igualdad del módulo. $\overline{M} = M_{\text{term}} \in \Delta$ por `M_term_eq_natToTerm`.

Paso 5: Congruencia de la multiplicación. Usando el paso 4 y reflexividad de $\overline{Q}$: $\overline{Q} \cdot \overline{M} = \overline{Q} \cdot M_{\text{term}} \in \Delta$ por `eq_mul_Δ`.

Paso 6: Transitividad con el paso 3. $\overline{Q \cdot M} = \overline{Q} \cdot M_{\text{term}} \in \Delta$.

Paso 7: Congruencia de la adición. Combinando el paso 6 con reflexividad de $\overline{R}$: $\overline{Q \cdot M} + \overline{R} = \overline{Q} \cdot M_{\text{term}} + \overline{R} \in \Delta$.

Paso 8: Transitividad con el paso 2. $\overline{Q \cdot M + R} = \overline{Q} \cdot M_{\text{term}} + \overline{R} \in \Delta$.

Paso 9: Transitividad final con el paso 1. $\overline{A} = \overline{Q} \cdot M_{\text{term}} + \overline{R} \in \Delta$.

Lema `lt_form_M_term_to_natToTerm`: Conversión entre fórmulas de orden

Establece que si un término cerrado R' satisface la fórmula de orden con el término compuesto M_{term} (usando variable testigo 5), entonces también satisface la fórmula de orden con el numeral $\overline{M}$ (usando variable testigo 1):

$$ \Delta \vdash R' < M_{\text{term}} \text{ (con testigo 5)} \implies \Delta \vdash R' < \overline{M} \text{ (con testigo 1)} $$

Estructura de la demostración

1. **Extracción del testigo:** De la hipótesis $R' < M_{\text{term}}$ (con variable testigo 5), se tiene $\exists v_5\ (M_{\text{term}} = R' + S(v_5)) \in \Delta$. Por la propiedad de Henkin, existe un testigo cerrado d tal que $M_{\text{term}} = R' + S(d) \in \Delta$.

2. **Simplificación de la sustitución:** La instancia con $v_5 := d$ se simplifica usando que M_{term} y R' son términos cerrados, por lo que la sustitución solo afecta a la variable v_5, reemplazándola por d.

3. **Igualdad del módulo:** Por `M_term_eq_natToTerm`, se tiene $\overline{M} = M_{\text{term}} \in \Delta$. Por simetría y transitividad con el paso anterior: $\overline{M} = R' + S(d) \in \Delta$.

4. **Reintroducción del existencial con variable 1:** Se sustituye v_1 por d en la fórmula $\overline{M} = R' + S(v_1)$. La simplificación es directa porque $\overline{M}$ y R' son cerrados. Se aplica introducción del existencial (`exists_intro`) con testigo d, obteniendo $\exists v_1\ (\overline{M} = R' + S(v_1)) \in \Delta$.

5. **Conversión a `lt_form`:** Por definición, $\exists v_1\ (\overline{M} = R' + S(v_1))$ es exactamente `substForm 0 R' (lt_form (var 0) (natToTerm M) 1)`, ya que la sustitución de v_0 por R' en la fórmula de orden con variable testigo 1 produce precisamente esa expresión (lema `substForm_lt_form`).

Significado para la función beta

Estos dos lemas proporcionan los puentes necesarios entre las diferentes partes de la fórmula `BetaForm`. La fórmula `BetaForm` expresa internamente la relación:

$$ \exists Q, R\ (A = Q \cdot M_{\text{term}} + R \land R < M_{\text{term}}) $$

Sin embargo, para utilizar las propiedades ya demostradas sobre `lt_form` y las operaciones aritméticas (que están formuladas en términos de numerales $\overline{M}$), es necesario poder traducir entre la representación con M_{term} y la representación con $\overline{M}$. Los lemas `eq_A_QMR_term` y `lt_form_M_term_to_natToTerm` realizan exactamente esta traducción, garantizando que ambas formulaciones son equivalentes en Δ. Esto será crucial en la demostración final de `beta_representable`, donde se deberá probar simultáneamente la condición de gráfico (existencia de Q y R) y la condición de funcionalidad (unicidad de R).

/-- eq A (add (mul Q M_term) R) ∈ Δ -/

lemma eq_A_QMR_term {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

(h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

(A B I R Q : Nat) (h_div : A = Q * (1 + (I + 1) * B) + R) :

let _M := 1 + (I + 1) * B

let M_term := ArithTerm.succ (ArithTerm.mul (ArithTerm.succ (natToTerm (s := s) I)) (natToTerm (s := s) B))

ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (natToTerm (s := s) Q) M_term) (natToTerm (s := s) R)) ∈ Δ := by

intro M M_term

```
  -- Paso 1: natToTerm A = natToTerm (Q * M + R) por h_div

  have h_A_rw : ArithForm.eq (natToTerm (s := s) A) (natToTerm (s := s) (Q * M + R)) ∈
Δ := by

    rw [h_div]

    exact eq_refl_in_Δ Δ h_max h_con _

  -- Paso 2: natToTerm (Q * M + R) = add (natToTerm (Q * M)) (natToTerm R)

  have h_add_hom := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ (Q *
M) R

  -- Paso 3: natToTerm (Q * M) = mul (natToTerm Q) (natToTerm M)

  have h_mul_hom := natToTerm_mul_hom Δ h_max h_con h_add_zero h_add_succ
h_mul_zero h_mul_succ Q M

  -- Paso 4: natToTerm M = M_term

  have h_M_eq := M_term_eq_natToTerm Δ h_max h_con h_add_zero h_add_succ
h_mul_zero h_mul_succ I B

  -- Paso 5: mul (natToTerm Q) (natToTerm M) = mul (natToTerm Q) M_term
(congruencia de mul)

  have h_refl_Q := eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) Q)

  have h_mul_congr : ArithForm.eq (ArithTerm.mul (natToTerm (s := s) Q) (natToTerm
(s := s) M))

                                  (ArithTerm.mul (natToTerm (s := s) Q) M_term) ∈ Δ :=

    maximal_consistent_closed_under_deduction Δ h_max h_con _

      (SPA_Deduction_Internal.eq_mul Δ _ _ _ _

        (SPA_Deduction_Internal.hyp Δ _ h_refl_Q)

        (SPA_Deduction_Internal.hyp Δ _ h_M_eq))

  -- Paso 6: natToTerm (Q * M) = mul (natToTerm Q) M_term (transitividad)

  have h_mul_trans := eq_trans_Δ h_max h_con h_mul_hom h_mul_congr

  -- Paso 7: add (natToTerm (Q*M)) (natToTerm R) = add (mul (natToTerm Q)
M_term) (natToTerm R)

  have h_refl_R := eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) R)

  have h_add_congr := eq_add_Δ h_max h_con h_mul_trans h_refl_R

  -- Paso 8: natToTerm (Q*M+R) = add (mul (natToTerm Q) M_term) (natToTerm R)
```

have h_add_trans := eq_trans_Δ h_max h_con h_add_hom h_add_congr

-- Paso 9: natToTerm A = add (mul (natToTerm Q) M_term) (natToTerm R)

exact eq_trans_Δ h_max h_con h_A_rw h_add_trans

/-- Conversión de lt_form R' M_term 5 a substForm 0 R'.val (lt_form (var 0) (natToTerm M) 1) -/

lemma lt_form_M_term_to_natToTerm {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

(h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

(I B : Nat) (R' : TermClosed s)

(h_lt : lt_form R'.val (ArithTerm.succ (ArithTerm.mul (ArithTerm.succ (natToTerm (s := s) I)) (natToTerm (s := s) B))) 5 ∈ Δ) :

let M := 1 + (I + 1) * B

substForm 0 R'.val (lt_form (ArithTerm.var 0) (natToTerm (s := s) M) 1) ∈ Δ := by

intro M

let M_term : ArithTerm s := ArithTerm.succ (ArithTerm.mul (ArithTerm.succ (natToTerm (s := s) I)) (natToTerm (s := s) B))

-- De h_lt: $\exists_5$ (eq M_term (add R'.val (succ (var 5)))) ∈ Δ

have h_exists_in : exists_form 5 (ArithForm.eq M_term (ArithTerm.add R'.val (ArithTerm.succ (ArithTerm.var 5)))) ∈ Δ := h_lt

-- Por Henkin: existe d tal que eq M_term (add R'.val (succ d.val)) ∈ Δ

obtain ⟨d, hd⟩ := h_henkin 5 (ArithForm.eq M_term (ArithTerm.add R'.val (ArithTerm.succ (ArithTerm.var 5)))) h_exists_in

-- Simplificar hd

have hd_simp : substForm 5 d.val (ArithForm.eq M_term (ArithTerm.add R'.val (ArithTerm.succ (ArithTerm.var 5)))) =

ArithForm.eq M_term (ArithTerm.add R'.val (ArithTerm.succ d.val)) := by

dsimp [substForm, substTerm]

have h_M_fresh := substTerm_fresh M_term 5 d.val (NotOccur_of_isClosed M_term (by

dsimp [M_term, isClosed]; rw [isClosed_natToTerm, isClosed_natToTerm]; rfl) 5)

have h_R_fresh := substTerm_fresh R'.val 5 d.val (NotOccur_of_isClosed R'.val R'.property 5)

rw [h_M_fresh, h_R_fresh]

rw [hd_simp] at hd

-- M_term = natToTerm M en Δ

have h_M_eq := M_term_eq_natToTerm Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ I B

-- natToTerm M = M_term en Δ (simetría)

have h_M_eq_symm := eq_symm_Δ h_max h_con h_M_eq

-- natToTerm M = add R'.val (succ d.val) en Δ

have h_natToTerm_M_eq := eq_trans_Δ h_max h_con h_M_eq hd

-- exists_intro con testigo d.val para la variable 1

have h_subst : substForm 1 d.val (ArithForm.eq (natToTerm (s := s) M) (ArithTerm.add R'.val (ArithTerm.succ (ArithTerm.var 1)))) =

ArithForm.eq (natToTerm (s := s) M) (ArithTerm.add R'.val (ArithTerm.succ d.val)) := by

dsimp [substForm, substTerm]

have h_M_fresh := substTerm_fresh (natToTerm (s := s) M) 1 d.val (NotOccur_of_isClosed (natToTerm (s := s) M) (isClosed_natToTerm M) 1)

have h_R_fresh := substTerm_fresh R'.val 1 d.val (NotOccur_of_isClosed R'.val R'.property 1)

rw [h_M_fresh, h_R_fresh]

have h_subst_in : substForm 1 d.val (ArithForm.eq (natToTerm (s := s) M) (ArithTerm.add R'.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ := by

rwa [h_subst]

have h_free : IsFreeFor 1 d.val (ArithForm.eq (natToTerm (s := s) M) (ArithTerm.add R'.val (ArithTerm.succ (ArithTerm.var 1)))) :=

```
  IsFreeFor_closed _ 1 d

 have h_exists := exists_intro Δ 1 (ArithForm.eq (natToTerm (s := s) M) (ArithTerm.add R'.val (ArithTerm.succ (ArithTerm.var 1)))) d.val h_free (SPA_Deduction_Internal.hyp Δ _ h_subst_in)

 have h_exists_in' : exists_form 1 (ArithForm.eq (natToTerm (s := s) M) (ArithTerm.add R'.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ :=

  maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists

 -- Convertir a substForm 0 R'.val (lt_form (var 0) (natToTerm M) 1)

 rw [substForm_lt_form R' (natToTerm (s := s) M) (isClosed_natToTerm M)]

 exact h_exists_in'
```

Lema `add_assoc_Δ`: Asociatividad general de la adición en Δ

Se demuestra la propiedad asociativa de la adición para términos cerrados arbitrarios dentro de Δ. A diferencia del lema `add_assoc_numeral_Δ` (que requería que el tercer término fuera un numeral), esta versión es completamente general:

$$ \forall a, b, c \in \text{TermClosed } s,\ \Delta \vdash (a + b) + c = a + (b + c) $$

Estructura de la demostración

La prueba sigue el patrón de inducción interna sobre el tercer término c, generalizando el lema `add_assoc_numeral_Δ` a términos cerrados arbitrarios.

1. Variable fresca y fórmula de inducción: Se elige $v := \max_var_term(c.\text{val}) + 1$, fresca respecto a c. La fórmula de inducción es:

$$ \varphi'(v) := \text{Ax0}_{a+b} \to (\text{Ax0}_b \to (\forall v\ \text{Ax1}_{a+b}(v)) \to (\forall v\ \text{Ax1}_b(v)) \to (\forall v\ \text{Ax1}_a(v)) \to \text{Eq}_v) $$

donde:

- $\text{Ax0}_{a+b} = ((a + b) + 0 = a + b)$

- $\text{Ax0}_b = (b + 0 = b)$

- $\forall v\ \text{Ax1}_{a+b}(v) = \forall v\ ((a + b) + S(v) = S((a + b) + v))$

- $\forall v\ \text{Ax1}_b(v) = \forall v\ (b + S(v) = S(b + v))$

- $\forall v\ \text{Ax1}_a(v) = \forall v\ (a + S(b + v) = S(a + (b + v)))$

- $\text{Eq}_v = ((a + b) + v = a + (b + v))$

Esta fórmula codifica la asociatividad como una consecuencia de los axiomas de la adición, donde $\forall v\ \text{Ax1}_a(v)$ es una versión modificada del axioma `add_succ` que incorpora la estructura asociativa en el segundo argumento.

2. Caso base ($v := 0$): Se debe probar $(a + b) + 0 = a + (b + 0)$ bajo los axiomas. La demostración usa:

- Ax0_{a+b}: $(a + b) + 0 = a + b$

- Ax0_b: $b + 0 = b$, de donde $a + (b + 0) = a + b$ por congruencia

- Transitividad y simetría para conectar ambas igualdades.

3. Paso inductivo ($v \to S(v)$): Asumiendo $\varphi'(v)$ (que da $(a + b) + v = a + (b + v)$) y todos los axiomas, se debe probar $(a + b) + S(v) = a + (b + S(v))$. La cadena de igualdades es:

- Por $\text{Ax1}_{a+b}(v)$: $(a + b) + S(v) = S((a + b) + v)$

- Por hipótesis inductiva y congruencia: $S((a + b) + v) = S(a + (b + v))$

- Por $\text{Ax1}_b(v)$ y congruencia con a: $a + (b + S(v)) = a + S(b + v)$

- Por $\text{Ax1}_a(v)$: $a + S(b + v) = S(a + (b + v))$

- Por transitividad y simetría se conectan ambas ramas.

4. Instanciación y descarga de axiomas: La inducción proporciona $\forall v\ \varphi'(v)$. Instanciando con $c.\text{val}$ (verificando libertad de sustitución) se obtiene $\varphi'(c) \in \Delta$. Las cinco premisas se descargan:

- Ax0_{a+b} y Ax0_b están en Δ por `h_add_zero`.

- Las tres fórmulas cuantificadas universalmente se demuestra que pertenecen a Δ usando maximalidad y Henkin: si alguna no perteneciera, su negación estaría en Δ, y por Henkin existiría un contraejemplo que contradice `h_add_succ` (para las dos primeras) o una instancia adecuada de `h_add_succ` (para la tercera).

- Tras cinco eliminaciones de la implicación, se obtiene $(a + b) + c = a + (b + c) \in \Delta$.

Significado

`add_assoc_Δ` completa el desarrollo de las propiedades algebraicas de la adición en Δ, estableciendo que los términos cerrados forman un semigrupo conmutativo (con `add_comm_Δ`) y asociativo respecto a la igualdad demostrable. La demostración ilustra la potencia del método de inducción interna: lo que en el modelo estándar es una consecuencia inmediata de la asociatividad de la adición en $\mathbb{N}$, en el sistema formal requiere una construcción cuidadosa que combine los axiomas de la adición, el principio de inducción, y la maquinaria de maximalidad y Henkin para garantizar la validez universal de los axiomas. Este lema será utilizado en la demostración de `beta_representable` para manipular expresiones algebraicas que involucran sumas anidadas, como las que aparecen en la igualdad $A = Q \cdot M + R$ y en las propiedades de la función beta.

-- Asociatividad de la adición en Δ -/

lemma add_assoc_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(a b c : TermClosed s) :

ArithForm.eq (ArithTerm.add (ArithTerm.add a.val b.val) c.val) (ArithTerm.add a.val (ArithTerm.add b.val c.val)) ∈ Δ := by

let v := max_var_term c.val + 1

have hv : NotOccur v c.val := NotOccur_above_max_var_term c.val v (by omega)

let Ax0_ab : ArithForm s := add_zero_axiom (ArithTerm.add a.val b.val)

let Ax0_b : ArithForm s := add_zero_axiom b.val

let Ax1_ab_body : ArithForm s := add_succ_axiom (ArithTerm.add a.val b.val) (ArithTerm.var v)

let Ax1_ab_forall : ArithForm s := ArithForm.forall_form v Ax1_ab_body

let Ax1_b_body : ArithForm s := add_succ_axiom b.val (ArithTerm.var v)

let Ax1_b_forall : ArithForm s := ArithForm.forall_form v Ax1_b_body

let Ax1_a_body : ArithForm s := add_succ_axiom a.val (ArithTerm.add b.val (ArithTerm.var v))

let Ax1_a_forall : ArithForm s := ArithForm.forall_form v Ax1_a_body

let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.add (ArithTerm.add a.val b.val) (ArithTerm.var v))

(ArithTerm.add a.val (ArithTerm.add b.val (ArithTerm.var v)))

let φ' : ArithForm s := ArithForm.impl Ax0_ab (ArithForm.impl Ax0_b (ArithForm.impl Ax1_ab_forall (ArithForm.impl Ax1_b_forall (ArithForm.impl Ax1_a_forall Eq_v))))

```
  have h_subst_phi' : ∀ u : ArithTerm s, substForm v u φ' =

    ArithForm.impl Ax0_ab (ArithForm.impl Ax0_b (ArithForm.impl Ax1_ab_forall (ArithForm.impl Ax1_b_forall (ArithForm.impl Ax1_a_forall

      (ArithForm.eq (ArithTerm.add (ArithTerm.add a.val b.val) u) (ArithTerm.add a.val (ArithTerm.add b.val u)))))) := by

    intro u

    dsimp [φ', Ax0_ab, Ax0_b, Ax1_ab_forall, Ax1_b_forall, Ax1_a_forall, Eq_v, add_zero_axiom, substForm, substTerm]

    have ha := substTerm_fresh a.val v u (NotOccur_of_isClosed a.val a.property v)

    have hb := substTerm_fresh b.val v u (NotOccur_of_isClosed b.val b.property v)

    rw [ha, hb]

    have h_if : (if v = v then u else ArithTerm.var v) = u := if_pos rfl

    rw [h_if]

    have h_if1 : (if v = v then ArithForm.forall_form v Ax1_ab_body else ArithForm.forall_form v (substForm v u Ax1_ab_body)) = ArithForm.forall_form v Ax1_ab_body := if_pos rfl

    rw [h_if1]

    have h_if2 : (if v = v then ArithForm.forall_form v Ax1_b_body else ArithForm.forall_form v (substForm v u Ax1_b_body)) = ArithForm.forall_form v Ax1_b_body := if_pos rfl

    rw [h_if2]

    have h_if3 : (if v = v then ArithForm.forall_form v Ax1_a_body else ArithForm.forall_form v (substForm v u Ax1_a_body)) = ArithForm.forall_form v Ax1_a_body := if_pos rfl

    rw [h_if3]

  -- Caso base

  have h_base : SPA_Deduction_Internal s ∅ (substForm v ArithTerm.zero φ') := by

    rw [h_subst_phi' ArithTerm.zero]

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro
```

```
  apply SPA_Deduction_Internal.imp_intro

  let Γ5 : Set (ArithForm s) := insert Ax1_a_forall (insert Ax1_b_forall (insert Ax1_ab_forall (insert Ax0_b (insert Ax0_ab ∅))))

  have h_Ax0_ab : SPA_Deduction_Internal s Γ5 Ax0_ab := SPA_Deduction_Internal.hyp Γ5 Ax0_ab (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))

  have h_Ax0_b : SPA_Deduction_Internal s Γ5 Ax0_b := SPA_Deduction_Internal.hyp Γ5 Ax0_b (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

  have h_refl_a : SPA_Deduction_Internal s Γ5 (ArithForm.eq a.val a.val) := SPA_Deduction_Internal.eq_refl Γ5 a.val

  have h_congr : SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.add a.val (ArithTerm.add b.val ArithTerm.zero)) (ArithTerm.add a.val b.val)) :=

    SPA_Deduction_Internal.eq_add Γ5 _ _ _ _ h_refl_a (SPA_Deduction_Internal.hyp Γ5 Ax0_b (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))

  have h_symm : SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.add a.val b.val) (ArithTerm.add a.val (ArithTerm.add b.val ArithTerm.zero))) :=

    SPA_Deduction_Internal.eq_symm Γ5 _ _ h_congr

  exact SPA_Deduction_Internal.eq_trans Γ5 _ _ _ (SPA_Deduction_Internal.hyp Γ5 Ax0_ab (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))) h_symm

 -- Paso inductivo

 have h_step : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v (ArithForm.impl φ' (substForm v (ArithTerm.succ (ArithTerm.var v)) φ'))) := by

  apply SPA_Deduction_Internal.forall_intro

  · rw [h_subst_phi' (ArithTerm.succ (ArithTerm.var v))]

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

  let Γ6 : Set (ArithForm s) := insert Ax1_a_forall (insert Ax1_b_forall (insert Ax1_ab_forall (insert Ax0_b (insert Ax0_ab (insert φ' ∅)))))
```

have h_φ' : SPA_Deduction_Internal s Γ6 φ' := SPA_Deduction_Internal.hyp Γ6 φ' (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl))))))

have h_Ax0_ab : SPA_Deduction_Internal s Γ6 Ax0_ab := SPA_Deduction_Internal.hyp Γ6 Ax0_ab (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))

have h_Ax0_b : SPA_Deduction_Internal s Γ6 Ax0_b := SPA_Deduction_Internal.hyp Γ6 Ax0_b (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

have h_Ax1_ab : SPA_Deduction_Internal s Γ6 Ax1_ab_forall := SPA_Deduction_Internal.hyp Γ6 Ax1_ab_forall (Or.inr (Or.inr (Or.inl rfl)))

have h_Ax1_b : SPA_Deduction_Internal s Γ6 Ax1_b_forall := SPA_Deduction_Internal.hyp Γ6 Ax1_b_forall (Or.inr (Or.inl rfl))

have h_Ax1_a : SPA_Deduction_Internal s Γ6 Ax1_a_forall := SPA_Deduction_Internal.hyp Γ6 Ax1_a_forall (Or.inl rfl)

have h_imp1 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_φ' h_Ax0_ab

have h_imp2 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp1 h_Ax0_b

have h_imp3 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp2 h_Ax1_ab

have h_imp4 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp3 h_Ax1_b

have h_Eq_v := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp4 h_Ax1_a

-- Ax1_ab: add(add(a,b), S(v)) = S(add(add(a,b), v))

have h_ax1_ab_v := SPA_Deduction_Internal.forall_elim Γ6 v Ax1_ab_body (ArithTerm.var v) h_Ax1_ab True.intro

dsimp [Ax1_ab_body, add_succ_axiom, substForm, substTerm] at h_ax1_ab_v

have ha_f := substTerm_fresh a.val v (ArithTerm.var v) (NotOccur_of_isClosed a.val a.property v)

have hb_f := substTerm_fresh b.val v (ArithTerm.var v) (NotOccur_of_isClosed b.val b.property v)

rw [ha_f, hb_f] at h_ax1_ab_v

have h_vv : (if v = v then (ArithTerm.var v : ArithTerm s) else ArithTerm.var v) = ArithTerm.var v := if_pos rfl

rw [h_vv] at h_ax1_ab_v

-- S(add(add(a,b),v)) = S(add(a,add(b,v)))

have h_succ_hyp := SPA_Deduction_Internal.eq_succ Γ6 _ _ h_Eq_v

-- Ax1_b: add(b, S(v)) = S(add(b, v))

```
    have h_ax1_b_v := SPA_Deduction_Internal.forall_elim Γ6 v Ax1_b_body (ArithTerm.var v) h_Ax1_b True.intro
    dsimp [Ax1_b_body, add_succ_axiom, substForm, substTerm] at h_ax1_b_v
    rw [hb_f] at h_ax1_b_v
    rw [h_vv] at h_ax1_b_v
    -- add(a, add(b, S(v))) = add(a, S(add(b, v)))
    have h_refl_a : SPA_Deduction_Internal s Γ6 (ArithForm.eq a.val a.val) := SPA_Deduction_Internal.eq_refl Γ6 a.val
    have h_congr1 := SPA_Deduction_Internal.eq_add Γ6 _ _ _ _ h_refl_a h_ax1_b_v
    -- Ax1_a: add(a, S(add(b, v))) = S(add(a, add(b, v)))
    have h_ax1_a_v := SPA_Deduction_Internal.forall_elim Γ6 v Ax1_a_body (ArithTerm.var v) h_Ax1_a True.intro
    dsimp [Ax1_a_body, add_succ_axiom, substForm, substTerm] at h_ax1_a_v
    rw [ha_f, hb_f] at h_ax1_a_v
    rw [h_vv] at h_ax1_a_v
    -- Cadenas
    have h_lhs := SPA_Deduction_Internal.eq_trans Γ6 _ _ _ h_ax1_ab_v h_succ_hyp
    have h_rhs := SPA_Deduction_Internal.eq_trans Γ6 _ _ _ h_congr1 h_ax1_a_v
    have h_rhs_symm := SPA_Deduction_Internal.eq_symm Γ6 _ _ h_rhs
    exact SPA_Deduction_Internal.eq_trans Γ6 _ _ _ h_lhs h_rhs_symm
  · intro ψ hψ
    nomatch hψ
-- Inducción
have h_ind : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ') :=
  SPA_Deduction_Internal.induction ∅ φ' v h_base h_step
have h_ind_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form v φ') :=
  SPA_Deduction_Internal.weaken ∅ _ Δ h_ind (Set.empty_subset _)
-- Instanciar con c.val
have h_free : IsFreeFor v c.val φ' := by
```

```
  dsimp [φ', Ax0_ab, Ax0_b, Ax1_ab_forall, Ax1_b_forall, Ax1_a_forall, Eq_v,
add_zero_axiom, add_succ_axiom, IsFreeFor]

  have h1 : (if v = v then True else NotOccur v c.val ∧ IsFreeFor v c.val Ax1_ab_body) =
True := if_pos rfl

  have h2 : (if v = v then True else NotOccur v c.val ∧ IsFreeFor v c.val Ax1_b_body) =
True := if_pos rfl

  have h3 : (if v = v then True else NotOccur v c.val ∧ IsFreeFor v c.val Ax1_a_body) =
True := if_pos rfl

  rw [h1, h2, h3]

  exact ⟨True.intro, True.intro, True.intro, True.intro, True.intro, True.intro⟩

 have h_elim := SPA_Deduction_Internal.forall_elim Δ v φ' c.val h_ind_Δ h_free

 rw [h_subst_phi' c.val] at h_elim

 -- Eliminar implicaciones

 have h_Ax0_ab_Δ : SPA_Deduction_Internal s Δ Ax0_ab := SPA_Deduction_Internal.hyp
Δ _ (h_add_zero (ArithTerm.add a.val b.val))

 have h_Ax0_b_Δ : SPA_Deduction_Internal s Δ Ax0_b := SPA_Deduction_Internal.hyp Δ
_ (h_add_zero b.val)

 have h_elim1 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim h_Ax0_ab_Δ

 have h_elim2 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim1 h_Ax0_b_Δ

 -- Ax1_ab_forall ∈ Δ

 have h_Ax1_ab_in : Ax1_ab_forall ∈ Δ := by

  by_contra h_not

  have h_neg : ArithForm.neg Ax1_ab_forall ∈ Δ := by cases h_max Ax1_ab_forall with |
inl h => contradiction | inr h => exact h

  have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg
Ax1_ab_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]

  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_ab_body) h_nd

  dsimp [Ax1_ab_body, add_succ_axiom, substForm, substTerm] at ht

  rw [substTerm_fresh a.val v t.val (NotOccur_of_isClosed a.val a.property v),

    substTerm_fresh b.val v t.val (NotOccur_of_isClosed b.val b.property v)] at ht

  have h_if_t : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
```

```
    rw [h_if_t] at ht
    have h_pos := h_add_succ (ArithTerm.add a.val b.val) t.val
    exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht))
  -- Ax1_b_forall ∈ Δ
  have h_Ax1_b_in : Ax1_b_forall ∈ Δ := by
    by_contra h_not
    have h_neg : ArithForm.neg Ax1_b_forall ∈ Δ := by cases h_max Ax1_b_forall with | inl h => contradiction | inr h => exact h
    have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_b_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
    obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_b_body) h_nd
    dsimp [Ax1_b_body, add_succ_axiom, substForm, substTerm] at ht
    rw [substTerm_fresh b.val v t.val (NotOccur_of_isClosed b.val b.property v)] at ht
    have h_if_t : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
    rw [h_if_t] at ht
    have h_pos := h_add_succ b.val t.val
    exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht))
  -- Ax1_a_forall ∈ Δ
  have h_Ax1_a_in : Ax1_a_forall ∈ Δ := by
    by_contra h_not
    have h_neg : ArithForm.neg Ax1_a_forall ∈ Δ := by cases h_max Ax1_a_forall with | inl h => contradiction | inr h => exact h
    have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_a_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
    obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_a_body) h_nd
    dsimp [Ax1_a_body, add_succ_axiom, substForm, substTerm] at ht
    rw [substTerm_fresh a.val v t.val (NotOccur_of_isClosed a.val a.property v),
      substTerm_fresh b.val v t.val (NotOccur_of_isClosed b.val b.property v)] at ht
```

```
  have h_if_t : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
  rw [h_if_t] at ht
  have h_pos := h_add_succ a.val (ArithTerm.add b.val t.val)
  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht))
 have h_Ax1_ab_Δ : SPA_Deduction_Internal s Δ Ax1_ab_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_ab_in
 have h_Ax1_b_Δ : SPA_Deduction_Internal s Δ Ax1_b_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_b_in
 have h_Ax1_a_Δ : SPA_Deduction_Internal s Δ Ax1_a_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_a_in
 have h_elim3 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim2 h_Ax1_ab_Δ
 have h_elim4 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim3 h_Ax1_b_Δ
 have h_final := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim4 h_Ax1_a_Δ
 exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final
```

Lema `Q'_bounded`: Acotación del cociente en la división

Se demuestra una propiedad fundamental para la unicidad del residuo en la función beta: si un término cerrado Q' satisface la igualdad $\overline{A} = Q' \cdot \overline{M} + \overline{k}$ para algún k y $M \geq 1$, entonces Q' es estrictamente menor que $\overline{A + 1}$ internamente:

$$ \forall A, M, k \in \mathbb{N},\ \forall Q' \in \text{TermClosed } s,\ M \geq 1 \land \Delta \vdash \overline{A} = Q' \cdot \overline{M} + \overline{k} \implies \Delta \vdash Q' < \overline{A + 1} $$

Estructura de la demostración

La prueba es constructiva y se basa en una manipulación algebraica ingeniosa dentro de Δ para exhibir explícitamente un testigo d tal que $\overline{A + 1} = Q' + S(d)$.

1. Construcción del testigo: Se define $X := Q' \cdot \overline{M - 1}$ y $d := X + \overline{k}$. La idea es que d representa la diferencia entre $\overline{A + 1}$ y Q', es decir, $\overline{A + 1} = Q' + S(d)$.

2. Cadena de igualdades: Se construye una cadena de igualdades en Δ que transforma $\overline{A + 1}$ en $Q' + S(d)$:

- $\overline{A + 1} = S(\overline{A})$ (definición de `natToTerm`)

- $S(\overline{A}) = S(Q' \cdot \overline{M} + \overline{k})$ (por la hipótesis `h_eq` y congruencia)

- $S(Q' \cdot \overline{M} + \overline{k}) = Q' \cdot \overline{M} + S(\overline{k})$ (por el axioma `add_succ`)

- $\overline{M} = S(\overline{M - 1})$ (pues $M \geq 1$, por lo que $M = (M-1) + 1$)

- $Q' \cdot \overline{M} = Q' \cdot S(\overline{M - 1}) = Q' \cdot \overline{M - 1} + Q'$ (por el axioma `mul_succ`)

- Reagrupando: $Q' \cdot \overline{M} + S(\overline{k}) = (Q' \cdot \overline{M - 1} + Q') + S(\overline{k})$

- Por asociatividad y conmutatividad de la adición (usando `add_assoc_Δ` y `add_comm_Δ`): $(X + Q') + S(\overline{k}) = (X + S(\overline{k})) + Q' = Q' + (X + S(\overline{k})) = Q' + S(d)$.

3. Reordenamiento con lemas algebraicos: La demostración utiliza extensivamente los lemas de asociatividad (`add_assoc_Δ`), conmutatividad (`add_comm_Δ`) y congruencia (`eq_add_Δ`, `eq_succ_Δ`) para manipular las

expresiones. Cada paso de la cadena se justifica mediante uno de estos lemas o mediante los axiomas de la adición y multiplicación.

4. Introducción de la fórmula de orden: Una vez establecida la igualdad $\overline{A + 1} = Q' + S(d) \in \Delta$, se sustituye la variable testigo v_1 por d en la fórmula $\overline{A + 1} = Q' + S(v_1)$, se introduce el cuantificador existencial, y se aplica `substForm_lt_form` para reescribir el resultado como $Q' < \overline{A + 1}$.

Significado para la unicidad del residuo

Este lema es crucial para la demostración de la funcionalidad en `beta_representable`. Cuando se asume que existen dos posibles valores R y R' que satisfacen la relación de la función beta con el mismo A, B, I, pero posiblemente con distintos cocientes Q y Q', el lema `Q'_bounded` permite acotar Q' por $A + 1$. Esta acotación es necesaria para aplicar el principio de inducción o minimización que garantiza la unicidad del residuo. La construcción explícita del testigo d muestra que la acotación no es meramente existencial sino efectiva: el testigo se construye a partir de Q', M y k mediante operaciones aritméticas, lo cual es posible porque todos estos valores están representados como términos cerrados en el lenguaje objeto. La demostración ilustra de manera notable cómo las propiedades algebraicas demostradas previamente (asociatividad, conmutatividad, axiomas de la adición y multiplicación) se combinan para establecer hechos aritméticos no triviales dentro del sistema formal.

-- Q' < A + 1 en Δ -/

```
lemma Q'_bounded {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

  (_h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

  (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

  (A M k : Nat) (Q' : TermClosed s)

  (h_eq : ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul Q'.val (natToTerm (s := s) M)) (natToTerm (s := s) k)) ∈ Δ)

  (h_M_pos : M ≥ 1) :

  substForm 0 Q'.val (lt_form (ArithTerm.var 0) (natToTerm (s := s) (A + 1)) 1) ∈ Δ := by

 let X : ArithTerm s := ArithTerm.mul Q'.val (natToTerm (s := s) (M - 1))

 let d_val : ArithTerm s := ArithTerm.add X (natToTerm (s := s) k)

 have hX_closed : isClosed X = true := by dsimp [X, isClosed]; rw [Q'.property, isClosed_natToTerm]; rfl

 have hd_closed : isClosed d_val = true := by dsimp [d_val, isClosed]; rw [hX_closed, isClosed_natToTerm]; rfl

 let Xc : TermClosed s := ⟨X, hX_closed⟩

 let dc : TermClosed s := ⟨d_val, hd_closed⟩

 let skc : TermClosed s := ⟨ArithTerm.succ (natToTerm (s := s) k), by dsimp [isClosed]; rw [isClosed_natToTerm]⟩

 let Xskc : TermClosed s := ⟨ArithTerm.add X (ArithTerm.succ (natToTerm (s := s) k)), by dsimp [isClosed]; rw [hX_closed, isClosed_natToTerm]; rfl⟩

 have hntM_eq : ArithForm.eq (natToTerm (s := s) M) (ArithTerm.succ (natToTerm (s := s) (M - 1))) ∈ Δ := by

  cases M with

  | zero => omega

  | succ M' => apply eq_refl_in_Δ Δ h_max h_con

 have h1 : ArithForm.eq (natToTerm (s := s) (A + 1)) (ArithTerm.succ (natToTerm (s := s) A)) ∈ Δ :=

  eq_refl_in_Δ Δ h_max h_con _
```

have h2 := eq_succ_Δ h_max h_con h_eq

have h3 := eq_symm_Δ h_max h_con (h_add_succ (ArithTerm.mul Q'.val (natToTerm (s := s) M)) (natToTerm (s := s) k))

have hc1 := eq_trans_Δ h_max h_con h1 h2

have hc2 := eq_trans_Δ h_max h_con hc1 h3

have hmul_congr := eq_mul_Δ h_max h_con (eq_refl_in_Δ Δ h_max h_con Q'.val) hntM_eq

have hmul_succ_ax := h_mul_succ Q'.val (natToTerm (s := s) (M - 1))

have h4 := eq_trans_Δ h_max h_con hmul_congr hmul_succ_ax

have h5 := eq_add_Δ h_max h_con h4 (eq_refl_in_Δ Δ h_max h_con (ArithTerm.succ (natToTerm (s := s) k)))

have h6 := add_assoc_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ Xc Q' skc

have h7 := add_comm_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ Q'.val skc.val

have h8 := eq_add_Δ h_max h_con (eq_refl_in_Δ Δ h_max h_con X) h7

have h9 := eq_symm_Δ h_max h_con (add_assoc_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ Xc skc Q')

have h10 := add_comm_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ Xskc.val Q'.val

have h11 := eq_symm_Δ h_max h_con (h_add_succ X (natToTerm (s := s) k))

have h12 := eq_add_Δ h_max h_con (eq_refl_in_Δ Δ h_max h_con Q'.val) (eq_symm_Δ h_max h_con h11)

have hc3 := eq_trans_Δ h_max h_con hc2 h5

have hc4 := eq_trans_Δ h_max h_con hc3 h6

have hc5 := eq_trans_Δ h_max h_con hc4 h8

have hc6 := eq_trans_Δ h_max h_con hc5 h9

have hc7 := eq_trans_Δ h_max h_con hc6 h10

have hc8 := eq_trans_Δ h_max h_con hc7 h12

have hsub : substForm 1 d_val (ArithForm.eq (natToTerm (s := s) (A + 1)) (ArithTerm.add Q'.val (ArithTerm.succ (ArithTerm.var 1)))) =

    ArithForm.eq (natToTerm (s := s) (A + 1)) (ArithTerm.add Q'.val (ArithTerm.succ d_val)) := by

dsimp [substForm, substTerm]

rw [substTerm_fresh (natToTerm (s := s) (A + 1)) 1 d_val (NotOccur_of_isClosed _ (isClosed_natToTerm (A + 1)) 1),

substTerm_fresh Q'.val 1 d_val (NotOccur_of_isClosed _ Q'.property 1)]

have hsub_in : substForm 1 d_val (ArithForm.eq (natToTerm (s := s) (A + 1)) (ArithTerm.add Q'.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ := by

rwa [hsub]

have hfree : IsFreeFor 1 d_val (ArithForm.eq (natToTerm (s := s) (A + 1)) (ArithTerm.add Q'.val (ArithTerm.succ (ArithTerm.var 1)))) :=

IsFreeFor_closed _ 1 dc

have hex := exists_intro Δ 1 _ d_val hfree (SPA_Deduction_Internal.hyp Δ _ hsub_in)

have hex_in : exists_form 1 (ArithForm.eq (natToTerm (s := s) (A + 1)) (ArithTerm.add Q'.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ :=

maximal_consistent_closed_under_deduction Δ h_max h_con _ hex

rw [substForm_lt_form Q' (natToTerm (s := s) (A + 1)) (isClosed_natToTerm (A + 1))]

exact hex_in

Teorema principal `beta_representable`: Representabilidad de la función beta de Gödel

Se demuestra el teorema cumbre de esta sección: la función beta de Gödel es representable en cualquier teoría Δ maximal consistente con propiedad de Henkin que contenga los axiomas de la aritmética. Formalmente, para cualesquiera números naturales A, B, I, R tales que $R = \text{godelBeta}(A, B, I)$, la fórmula `BetaForm` representa correctamente esta relación:

$$ \forall A, B, I, R \in \mathbb{N},\ R = \beta(A, B, I) \implies $$

1. **Gráfico:** $\Delta \vdash \text{BetaForm}(\overline{A}, \overline{B}, \overline{I}, \overline{R})$

2. **Funcionalidad:** $\forall R' \in \text{TermClosed } s,\ \Delta \vdash \text{BetaForm}(\overline{A}, \overline{B}, \overline{I}, R') \implies \Delta \vdash R' = \overline{R}$

Estructura de la demostración

La prueba se organiza en dos partes principales, correspondientes a las dos condiciones de representabilidad. Se definen $M := 1 + (I + 1) \cdot B$ (el módulo de la división) y $M_{\text{term}} := S(S(\overline{I}) \cdot \overline{B})$ (su representación como término compuesto).

Parte 1: Condición de gráfico

Se debe demostrar que la instancia de `BetaForm` con los numerales correctos pertenece a Δ.

1. Cálculo del cociente: Sea $Q := A \mathbin{/\!/} M$ (división entera). Por el algoritmo de la división, se tiene $A = Q \cdot M + R$ con $R < M$.

2. Igualdad de la división: Se aplica el lema `eq_A_QMR_term` para obtener $\overline{A} = \overline{Q} \cdot M_{\text{term}} + \overline{R} \in \Delta$. Este lema traduce la igualdad numérica externa a una igualdad demostrable internamente usando el término compuesto M_{term}.

3. Desigualdad del residuo: Dado que $R < M$, se aplica `lt_form_M_term_intro` para obtener $\overline{R} < M_{\text{term}} \in \Delta$ (con variable testigo 5).

4. Conjunción y cuantificación existencial: Se combinan ambas fórmulas mediante `and_intro` para obtener la conjunción, y se introduce el cuantificador existencial sobre la variable v_4 con testigo $\overline{Q}$. La verificación de que la sustitución produce la forma requerida de `BetaForm` se realiza mediante una simplificación explícita que utiliza la clausura de los numerales para eliminar las sustituciones que no tienen efecto.

Parte 2: Condición de funcionalidad

Se asume que para algún término cerrado R', se tiene $\text{BetaForm}(\overline{A}, \overline{B}, \overline{I}, R') \in \Delta$, y se debe demostrar $R' = \overline{R} \in \Delta$.

1. Extracción del testigo: La hipótesis es una fórmula existencial $\exists v_4\ (\overline{A} = v_4 \cdot M_{\text{term}} + R' \land R' < M_{\text{term}})$. Por la propiedad de Henkin, existe un testigo cerrado Q' que satisface el cuerpo de la fórmula.

2. Simplificación del cuerpo: La instancia con $v_4 := Q'$ se simplifica (usando la clausura de M_{term}, R' y $\overline{A}$) para obtener:

$$ \overline{A} = Q' \cdot M_{\text{term}} + R' \in \Delta \quad \text{y} \quad R' < M_{\text{term}} \in \Delta $$

3. Conversión del orden: Por el lema `lt_form_M_term_to_natToTerm`, la desigualdad $R' < M_{\text{term}}$ se traduce a $R' < \overline{M} \in \Delta$.

4. Acotación del residuo: Por el lema `lt_numeral_bounded`, existe $k < M$ tal que $R' = \overline{k} \in \Delta$.

5. Sustitución de R' por $\overline{k}$: Se reemplaza R' por $\overline{k}$ en la igualdad de la división, obteniendo $\overline{A} = Q' \cdot M_{\text{term}} + \overline{k} \in \Delta$.

6. Conversión del módulo: Usando `M_term_eq_natToTerm` y congruencia, se reemplaza M_{term} por $\overline{M}$, obteniendo $\overline{A} = Q' \cdot \overline{M} + \overline{k} \in \Delta$.

7. Acotación del cociente: Por el lema `Q'_bounded`, se tiene $Q' < \overline{A + 1} \in \Delta$. Por `lt_numeral_bounded`, existe $q' < A + 1$ tal que $Q' = \overline{q'} \in \Delta$.

8. Sustitución de Q' por $\overline{q'}$: Se reemplaza en la igualdad, obteniendo $\overline{A} = \overline{q'} \cdot \overline{M} + \overline{k} \in \Delta$.

9. Traducción a igualdad numérica externa: Por los homomorfismos de la adición y multiplicación, $\overline{q' \cdot M + k} = \overline{q'} \cdot \overline{M} + \overline{k} \in \Delta$. Por transitividad y simetría, $\overline{A} = \overline{q' \cdot M + k} \in \Delta$. Dado que numerales distintos tienen negaciones demostrables (por `prove_distinct_numerals_Δ`), la consistencia de Δ fuerza la igualdad externa $A = q' \cdot M + k$.

10. Determinación de k: Por el algoritmo de la división, R es el único número menor que M que satisface $A = Q \cdot M + R$ para algún Q. Dado que $k < M$ y $A = q' \cdot M + k$, se concluye $k = R$. Por tanto, $R' = \overline{k} = \overline{R} \in \Delta$.

Significado del teorema

`beta_representable` es un resultado de importancia capital en la aritmetización de la sintaxis. La función beta de Gödel, definida como:

$$ \beta(a, b, i) = a \bmod (1 + (i + 1) \cdot b) $$

es la herramienta fundamental para codificar y decodificar secuencias finitas de números naturales. Gödel demostró que esta función es recursiva primitiva y que toda secuencia finita $(a_0, \ldots, a_{n-1})$ puede representarse mediante dos números a y b tales que $\beta(a, b, i) = a_i$ para todo $i < n$.

El teorema establece que esta función es representable en el sistema SPA, lo cual significa que la relación $r = \beta(a, b, i)$ puede expresarse dentro del lenguaje formal mediante la fórmula `BetaForm`. La demostración combina de manera magistral todos los ingredientes desarrollados a lo largo de esta fase:

- **Propiedades algebraicas** de la adición y multiplicación (asociatividad, conmutatividad, homomorfismos).

- **Propiedades de orden** (`lt_numeral_bounded`, `lt_form_intro`, `lt_form_M_term_intro`).

- **Propiedad de Henkin** para extraer testigos existenciales.

- **Maximalidad y consistencia** para cerrar deducciones y forzar igualdades externas.

- **Lemas de simplificación de sustituciones** para manipular fórmulas con múltiples cuantificadores.

La representabilidad de la función beta es el último eslabón antes de la representabilidad de todas las funciones recursivas primitivas, y constituye la base técnica sobre la que se construye la prueba de los teoremas de incompletitud de Gödel en el sistema SPA.

```
lemma beta_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con
: Consistent Δ) (h_henkin : HenkinSet Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

  (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

  (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

  (A B I R : Nat) (h_R : R = godelBeta A B I) :

  let _M := 1 + (I + 1) * B

  let _M_term := ArithTerm.succ (ArithTerm.mul (ArithTerm.succ (natToTerm (s := s)
I)) (natToTerm (s := s) B))

  (substForm 3 (natToTerm (s := s) R) (substForm 2 (natToTerm (s := s) I)
(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A)
BetaForm))) ∈ Δ) ∧

  (∀ (R' : TermClosed s),

   substForm 3 R'.val (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm
(s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm))) ∈ Δ →

   ArithForm.eq R'.val (natToTerm (s := s) R) ∈ Δ) := by

 intro M M_term

 have hM : M = 1 + (I + 1) * B := rfl

 have h_M_pos : M ≥ 1 := by omega

 have h_R_mod : R = A % M := by

  dsimp [godelBeta] at h_R

  rw [hM]

  exact h_R

 have h_R_lt : R < M := by

  rw [h_R_mod]

  exact Nat.mod_lt A (by omega)
```

```
 constructor
 · -- ===== GRÁFICO =====
   let Q := A / M
   have h_div_Q : A = Q * (1 + (I + 1) * B) + R := by
    dsimp [Q]
    rw [h_R_mod, hM]
    have h := Nat.div_add_mod A (1 + (I + 1) * B)
    rw [Nat.mul_comm] at h
    exact h.symm
   have h_eq_body := eq_A_QMR_term Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ A B I R Q h_div_Q
   have h_lt_body := lt_form_M_term_intro Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ R I B h_R_lt
   have h_and := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_eq_body) (SPA_Deduction_Internal.hyp Δ _ h_lt_body)
   have h_and_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and
   have h_subst_and : substForm 4 (natToTerm (s := s) Q) (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) M_term) (natToTerm (s := s) R))) (lt_form (natToTerm (s := s) R) M_term 5)) =
              and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (natToTerm (s := s) Q) M_term) (natToTerm (s := s) R))) (lt_form (natToTerm (s := s) R) M_term 5) := by
    dsimp [and_form, lt_form, exists_form, ArithForm.exists_form, substForm, substTerm]
    have h_A_fresh := substTerm_fresh (natToTerm (s := s) A) 4 (natToTerm (s := s) Q) (NotOccur_of_isClosed (natToTerm (s := s) A) (isClosed_natToTerm A) 4)
    have h_R_fresh := substTerm_fresh (natToTerm (s := s) R) 4 (natToTerm (s := s) Q) (NotOccur_of_isClosed (natToTerm (s := s) R) (isClosed_natToTerm R) 4)
    have h_M_fresh := substTerm_fresh M_term 4 (natToTerm (s := s) Q) (NotOccur_of_isClosed M_term (by
     dsimp [M_term, isClosed]; rw [isClosed_natToTerm, isClosed_natToTerm]; rfl) 4)
    rw [h_A_fresh, h_R_fresh, h_M_fresh]
```

```
  have h_subst_and_in : substForm 4 (natToTerm (s := s) Q) (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) M_term) (natToTerm (s := s) R))) (lt_form (natToTerm (s := s) R) M_term 5)) ∈ Δ := by

    rwa [h_subst_and]

  have h_free_Q : IsFreeFor 4 (natToTerm (s := s) Q) (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) M_term) (natToTerm (s := s) R))) (lt_form (natToTerm (s := s) R) M_term 5)) :=

    IsFreeFor_closed _ 4 ⟨natToTerm (s := s) Q, isClosed_natToTerm Q⟩

  have h_exists := exists_intro Δ 4 _ (natToTerm (s := s) Q) h_free_Q (SPA_Deduction_Internal.hyp Δ _ h_subst_and_in)

  have h_exists_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_exists

  have h_beta_subst : substForm 3 (natToTerm (s := s) R) (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm))) =

        exists_form 4 (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) M_term) (natToTerm (s := s) R))) (lt_form (natToTerm (s := s) R) M_term 5)) := by

    dsimp [BetaForm, M_term, exists_form, ArithForm.exists_form, and_form, ArithForm.and_form, lt_form, substForm, substTerm]

    have h_A0 := substTerm_fresh (natToTerm (s := s) A) 1 (natToTerm (s := s) B) (NotOccur_of_isClosed (natToTerm (s := s) A) (isClosed_natToTerm A) 1)

    have h_A1 := substTerm_fresh (natToTerm (s := s) A) 2 (natToTerm (s := s) I) (NotOccur_of_isClosed (natToTerm (s := s) A) (isClosed_natToTerm A) 2)

    have h_A2 := substTerm_fresh (natToTerm (s := s) A) 3 (natToTerm (s := s) R) (NotOccur_of_isClosed (natToTerm (s := s) A) (isClosed_natToTerm A) 3)

    have h_B0 := substTerm_fresh (natToTerm (s := s) B) 2 (natToTerm (s := s) I) (NotOccur_of_isClosed (natToTerm (s := s) B) (isClosed_natToTerm B) 2)

    have h_B1 := substTerm_fresh (natToTerm (s := s) B) 3 (natToTerm (s := s) R) (NotOccur_of_isClosed (natToTerm (s := s) B) (isClosed_natToTerm B) 3)

    have h_I0 := substTerm_fresh (natToTerm (s := s) I) 3 (natToTerm (s := s) R) (NotOccur_of_isClosed (natToTerm (s := s) I) (isClosed_natToTerm I) 3)

    rw [h_A0, h_A1, h_A2, h_B0, h_B1, h_I0]

  rw [h_beta_subst]

  exact h_exists_in
```

```
· -- ===== FUNCIONALIDAD =====

  intro R' h_in

  have h_beta_subst : substForm 3 R'.val (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm))) =

        exists_form 4 (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) M_term) R'.val)) (lt_form R'.val M_term 5)) := by

  dsimp [BetaForm, M_term, exists_form, ArithForm.exists_form, and_form, ArithForm.and_form, lt_form, substForm, substTerm]

  have h_A0 := substTerm_fresh (natToTerm (s := s) A) 1 (natToTerm (s := s) B) (NotOccur_of_isClosed (natToTerm (s := s) A) (isClosed_natToTerm A) 1)

  have h_A1 := substTerm_fresh (natToTerm (s := s) A) 2 (natToTerm (s := s) I) (NotOccur_of_isClosed (natToTerm (s := s) A) (isClosed_natToTerm A) 2)

  have h_A2 := substTerm_fresh (natToTerm (s := s) A) 3 R'.val (NotOccur_of_isClosed (natToTerm (s := s) A) (isClosed_natToTerm A) 3)

  have h_B0 := substTerm_fresh (natToTerm (s := s) B) 2 (natToTerm (s := s) I) (NotOccur_of_isClosed (natToTerm (s := s) B) (isClosed_natToTerm B) 2)

  have h_B1 := substTerm_fresh (natToTerm (s := s) B) 3 R'.val (NotOccur_of_isClosed (natToTerm (s := s) B) (isClosed_natToTerm B) 3)

  have h_I0 := substTerm_fresh (natToTerm (s := s) I) 3 R'.val (NotOccur_of_isClosed (natToTerm (s := s) I) (isClosed_natToTerm I) 3)

  rw [h_A0, h_A1, h_A2, h_B0, h_B1, h_I0]

 rw [h_beta_subst] at h_in

 obtain ⟨Q', hQ'⟩ := h_henkin 4 (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) M_term) R'.val)) (lt_form R'.val M_term 5)) h_in

 have hQ'_simp : substForm 4 Q'.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) M_term) R'.val)) (lt_form R'.val M_term 5)) =

       and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul Q'.val M_term) R'.val)) (lt_form R'.val M_term 5) := by

 dsimp [and_form, lt_form, exists_form, ArithForm.exists_form, substForm, substTerm]
```

```
    have h_A_fresh := substTerm_fresh (natToTerm (s := s) A) 4 Q'.val
(NotOccur_of_isClosed (natToTerm (s := s) A) (isClosed_natToTerm A) 4)

    have h_R_fresh := substTerm_fresh R'.val 4 Q'.val (NotOccur_of_isClosed R'.val
R'.property 4)

    have h_M_fresh := substTerm_fresh M_term 4 Q'.val (NotOccur_of_isClosed M_term
(by

      dsimp [M_term, isClosed]; rw [isClosed_natToTerm, isClosed_natToTerm]; rfl) 4)

    rw [h_A_fresh, h_R_fresh, h_M_fresh]

  rw [hQ'_simp] at hQ'

  have h_eq_comp := and_elim_left_Δ h_max h_con _ _ hQ'

  have h_lt_comp := and_elim_right_Δ h_max h_con _ _ hQ'

  have h_lt_natToTerm := lt_form_M_term_to_natToTerm Δ h_max h_con h_henkin
h_add_zero h_add_succ h_mul_zero h_mul_succ I B R' h_lt_comp

  obtain ⟨k, hk_lt, hk_eq⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero
h_add_succ M R' h_lt_natToTerm

  have h_eq_k_Mterm : ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add
(ArithTerm.mul Q'.val M_term) (natToTerm (s := s) k)) ∈ Δ := by

    have h_refl_QM := eq_refl_in_Δ Δ h_max h_con (ArithTerm.mul Q'.val M_term)

    have h_congr := eq_add_Δ h_max h_con h_refl_QM hk_eq

    exact eq_trans_Δ h_max h_con h_eq_comp h_congr

  have h_M_eq := M_term_eq_natToTerm Δ h_max h_con h_add_zero h_add_succ
h_mul_zero h_mul_succ I B

  have h_M_symm := eq_symm_Δ h_max h_con h_M_eq

  have h_congr_Q'M := eq_mul_Δ h_max h_con (eq_refl_in_Δ Δ h_max h_con Q'.val)
h_M_symm

  have h_congr_add_k := eq_add_Δ h_max h_con h_congr_Q'M (eq_refl_in_Δ Δ h_max
h_con (natToTerm (s := s) k))

  have h_eq_k : ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul
Q'.val (natToTerm (s := s) M)) (natToTerm (s := s) k)) ∈ Δ :=

    eq_trans_Δ h_max h_con h_eq_k_Mterm h_congr_add_k

  have h_Q'_bounded := Q'_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ
h_mul_zero h_mul_succ A M k Q' h_eq_k (by omega)
```

obtain ⟨q', hq'_lt, hq'_eq⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ (A + 1) Q' h_Q'_bounded

have h_eq_q'k_Mterm : ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (natToTerm (s := s) q') M_term) (natToTerm (s := s) k)) ∈ Δ := by

have h_congr_Q := eq_mul_Δ h_max h_con hq'_eq (eq_refl_in_Δ Δ h_max h_con M_term)

have h_congr := eq_add_Δ h_max h_con h_congr_Q (eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) k))

exact eq_trans_Δ h_max h_con h_eq_k_Mterm h_congr

have h_congr_M := eq_mul_Δ h_max h_con (eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) q')) h_M_symm

have h_congr_add := eq_add_Δ h_max h_con h_congr_M (eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) k))

have h_eq_q'k_M : ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (natToTerm (s := s) q') (natToTerm (s := s) M)) (natToTerm (s := s) k)) ∈ Δ :=

eq_trans_Δ h_max h_con h_eq_q'k_Mterm h_congr_add

have h_mul_hom := natToTerm_mul_hom Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ q' M

have h_add_hom := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ (q' * M) k

have h_congr_mul := eq_add_Δ h_max h_con h_mul_hom (eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) k))

have h_q'k_eq := eq_trans_Δ h_max h_con h_add_hom h_congr_mul

have h_A_q'k := eq_trans_Δ h_max h_con h_eq_q'k_M (eq_symm_Δ h_max h_con h_q'k_eq)

let Q := A / M

have h_A_QMR : ArithForm.eq (natToTerm (s := s) A) (natToTerm (s := s) (Q * M + R)) ∈ Δ := by

have h_QMR : Q * M + R = A := by

dsimp [Q]

rw [h_R_mod, hM]

have h := Nat.div_add_mod A (1 + (I + 1) * B)

```
      rw [Nat.mul_comm] at h
      exact h
    rw [h_QMR]
    exact eq_refl_in_Δ Δ h_max h_con _
  have h_QMR_q'k := eq_trans_Δ h_max h_con (eq_symm_Δ h_max h_con h_A_QMR) h_A_q'k
  have h_QMR_q'k_val : Q * M + R = q' * M + k := by
    by_contra h_neq
    have h_neg := prove_distinct_numerals_Δ h_max h_con (Q * M + R) (q' * M + k) h_neq
    have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
      SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_QMR_q'k) (SPA_Deduction_Internal.hyp Δ _ h_neg)
    exact h_con h_bot
  have h_k_eq_R : k = R := by
    have h_QMR_val : Q * M + R = A := by
      dsimp [Q]
      rw [h_R_mod, hM]
      have h := Nat.div_add_mod A (1 + (I + 1) * B)
      rw [Nat.mul_comm] at h
      exact h
    have h_q'k_val : q' * M + k = A := by
      omega
    have h_A_mod_R : A % M = R := by
      rw [h_R_mod]
    have h_A_mod_k : A % M = k := by
      rw [← h_q'k_val]
      have h1 : (k + q' * M) % M = k % M := by
        have h := Nat.add_mul_mod_self_left k M q'
```

```
      rw [Nat.mul_comm] at h
      exact h
    have h2 : k % M = k := Nat.mod_eq_of_lt hk_lt
    rw [Nat.add_comm]
    rw [h1, h2]
  omega
rw [h_k_eq_R] at hk_eq
exact hk_eq
```

Definiciones de `NumForm` y sus subfórmulas auxiliares

Se define la fórmula `NumForm` que representa la relación "z es el código del numeral de Peano correspondiente al número w". Esta es una de las fórmulas más complejas del desarrollo, ya que debe expresar internamente la construcción inductiva de los numerales mediante la función beta de Gödel y la codificación de términos.

`BetaForm_sh` y `PairForm_sh`: Versiones con variables ligadas desplazadas

Se definen versiones alpha-renombradas de `BetaForm` y `PairForm` que utilizan variables ligadas con índices más altos (8-9 y 10 respectivamente) para evitar colisiones con las variables utilizadas en `NumForm`:

$$ \text{BetaForm_sh} := \exists v_8\ (\overline{A} = v_8 \cdot M_{\text{term}} + \overline{R} \land \overline{R} < M_{\text{term}} \text{ (testigo 9)}) $$

$$ \text{PairForm_sh} := \text{versión de PairForm con variable ligada 10} $$

Intuición

El desplazamiento de variables ligadas es una técnica estándar para evitar capturas cuando se anidan múltiples cuantificadores. `NumForm` utiliza numerosas variables ligadas (hasta 13), por lo que es necesario renombrar las variables internas de `BetaForm` y `PairForm` para que no interfieran con las variables libres y ligadas del contexto.

`IsSuccCode`: Relación de códigos sucesores

Define la relación entre el código de un término v y el código de su sucesor $S(v)$:

$$ \text{IsSuccCode}(v, v') := \exists p\ (\text{PairForm_sh}(\overline{4}, v, p) \land v' = S(p)) $$

La etiqueta 4 corresponde al constructor `succ` en la serialización de términos (`serializeTerm`), por lo que el código de $S(t)$ es $\text{pair}(4, \text{código_de_}t) + 1$ (el $+1$ proviene del esquema de codificación de secuencias `encodeSeq`).

`NumForm`: Fórmula principal para la representación de numerales

$$ \text{NumForm}(w, z) := \exists a, b\ ($$

$$ \beta(a, b, 0) = \overline{\text{zeroCode}} \ \land $$

$$ \beta(a, b, w) = z \ \land $$

$$ \forall k\ (k < w \to \exists v, v'\ (\beta(a, b, k) = v \land \beta(a, b, k+1) = v' \land \text{IsSuccCode}(v, v')))) $$

Intuición y necesidad

Esta fórmula expresa que z es el código del numeral $\overline{w}$ mediante una codificación de la construcción inductiva:

1. **Caso base:** Existe una secuencia (codificada por a y b) cuyo primer elemento ($\beta(a, b, 0)$) es el código del término `zero`.

2. **Elemento final:** El w-ésimo elemento de la secuencia ($\beta(a, b, w)$) es z.

3. **Paso inductivo:** Para cada $k < w$, el $(k+1)$-ésimo elemento de la secuencia es el código del sucesor del k-ésimo elemento, es decir, la secuencia construye los códigos de los numerales paso a paso: $\text{zero}, S(\text{zero}), S(S(\text{zero})), \ldots$

Las variables ligadas utilizadas son:

- v_{12}, v_{13}: los parámetros a, b de la función beta

- v_4: variable de inducción k

- v_5, v_6: valores intermedios v, v'

- v_7: variable auxiliar para `IsSuccCode`

- v_{8}, v_9: variables internas de `BetaForm_sh`

- v_{10}: variable interna de `PairForm_sh`

- v_{11}: variable testigo para `lt_form`

Lemas de no ocurrencia para las subfórmulas

Se demuestra que las variables con índices distintos de los declarados como libres no ocurren libres en las subfórmulas, lo cual es indispensable para la manipulación correcta de sustituciones en `NumForm`.

`NotFreeForm_BetaForm_sh`

Establece que cualquier variable v distinta de $0, 1, 2, 3$ no es libre en `BetaForm_sh`. Las variables 0 a 3 son los argumentos A, B, I, R de la función beta; las variables 8 y 9 están ligadas internamente. La demostración verifica explícitamente cada subcomponente de la fórmula.

`NotFreeForm_PairForm_sh`

Establece que cualquier variable v distinta de $0, 1, 2$ no es libre en `PairForm_sh`. La variable 10 está ligada internamente.

`NotFreeForm_IsSuccCode`

Establece que cualquier variable v distinta de 5 y 6 no es libre en `IsSuccCode`. Las variables 5 y 6 son exactamente v y v' en la definición de `IsSuccCode`; la variable 7 está ligada.

Significado global

La construcción de `NumForm` representa un hito en la aritmetización de la sintaxis. Esta fórmula captura internamente la noción de "ser el código de un numeral", lo cual es un predicado no trivial porque los códigos se definen mediante serialización y codificación de secuencias. La definición utiliza la función beta para codificar la secuencia finita de códigos $\langle \text{código_de_zero}, \text{código_de_}S(\text{zero}), \ldots, \text{código_de_}S^w(\text{zero}) \rangle$, y `IsSuccCode` para expresar la relación entre códigos consecutivos. La verificación de las condiciones de no ocurrencia garantiza que las variables libres de `NumForm` son exactamente v_0 y v_1, como requiere la definición de representabilidad.

```
--
============================================================

-- ETAPA 1: DEFINICIONES DE NumForm

--
============================================================


/-- Código del término zero como Nat -/

def zeroCode : Nat := Encodable.encode (serializeTerm (ArithTerm.zero (s := 1)))


/-- Versión de BetaForm con vars internas 8,9 (alpha-renombre para evitar colisiones)
-/

def BetaForm_sh {s : Nat} : ArithForm s :=

  let a := ArithTerm.var 0

  let b := ArithTerm.var 1

  let i := ArithTerm.var 2

  let r := ArithTerm.var 3
```

```
 let M := ArithTerm.succ (ArithTerm.mul (ArithTerm.succ i) b)
 exists_form 8
  (and_form
   (ArithForm.eq a (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) M) r))
   (lt_form r M 9))

/-- Versión de PairForm con var interna 10 (alpha-renombre para evitar colisiones) -/
def PairForm_sh {s : Nat} : ArithForm s :=
 or_form
  (and_form (lt_form (ArithTerm.var 0) (ArithTerm.var 1) 10)
   (ArithForm.eq (ArithTerm.var 2) (ArithTerm.add (ArithTerm.mul (ArithTerm.var
1) (ArithTerm.var 1)) (ArithTerm.var 0))))
  (and_form (ArithForm.neg (lt_form (ArithTerm.var 0) (ArithTerm.var 1) 10))
   (ArithForm.eq (ArithTerm.var 2) (ArithTerm.add (ArithTerm.add (ArithTerm.mul
(ArithTerm.var 0) (ArithTerm.var 0)) (ArithTerm.var 0)) (ArithTerm.var 1))))

/-- IsSuccCode(v, v'): v' es el código del sucesor del término con código v.
  Es decir, v' = pair(4, v) + 1, donde 4 es la etiqueta de serializeTerm para succ. -/
def IsSuccCode {s : Nat} : ArithForm s :=
 exists_form 7 (and_form
  (substForm 2 (ArithTerm.var 7) (substForm 1 (ArithTerm.var 5) (substForm 0
(natToTerm (s := s) 4) PairForm_sh)))
  (ArithForm.eq (ArithTerm.var 6) (ArithTerm.succ (ArithTerm.var 7))))

/-- NumForm(w, z): z es el código del numeral de Peano de w.
  Variables libres: 0 (w), 1 (z).
  Variables ligadas: 2 (a), 3 (b), 4 (k), 5 (v), 6 (v'), 7 (p), 8-9 (BetaForm_sh), 10
(PairForm_sh), 11 (lt_form). -/
def NumForm {s : Nat} : ArithForm s :=
 exists_form 12 (exists_form 13 (
```

```
and_form
 -- β(a, b, 0) = ⌜ zero ⌝
 (substForm 3 (natToTerm (s := s) zeroCode)
  (substForm 2 (natToTerm (s := s) 0)
   (substForm 1 (ArithTerm.var 13)
    (substForm 0 (ArithTerm.var 12) BetaForm_sh))))
 (and_form
  -- β(a, b, w) = z
  (substForm 3 (ArithTerm.var 1)
   (substForm 2 (ArithTerm.var 0)
    (substForm 1 (ArithTerm.var 13)
     (substForm 0 (ArithTerm.var 12) BetaForm_sh))))
  -- ∀k (k < w → ∃v ∃v' (β(a,b,k) = v ∧ β(a,b,k+1) = v' ∧ IsSuccCode(v,v')))
  (ArithForm.forall_form 4 (ArithForm.impl
   (lt_form (ArithTerm.var 4) (ArithTerm.var 0) 11)
   (exists_form 5 (exists_form 6 (
    and_form
     -- β(a, b, k) = v
     (substForm 3 (ArithTerm.var 5)
      (substForm 2 (ArithTerm.var 4)
       (substForm 1 (ArithTerm.var 13)
        (substForm 0 (ArithTerm.var 12) BetaForm_sh))))
     (and_form
      -- β(a, b, k+1) = v'
      (substForm 3 (ArithTerm.var 6)
       (substForm 2 (ArithTerm.succ (ArithTerm.var 4))
        (substForm 1 (ArithTerm.var 13)
         (substForm 0 (ArithTerm.var 12) BetaForm_sh))))
```

```
            -- IsSuccCode(v, v')
            IsSuccCode
          )
        )))
      ))
    )
  ))
```

```
-- Las únicas variables libres de NumForm son 0 y 1 -/
```

```
lemma NotFreeForm_BetaForm_sh {s : Nat} (v : Nat)
    (hv0 : v ≠ 0) (hv1 : v ≠ 1) (hv2 : v ≠ 2) (hv3 : v ≠ 3) :
    NotFreeForm v (BetaForm_sh (s := s)) := by
  dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form,
NotFreeForm]
  by_cases hv8 : v = 8
  · rw [if_pos hv8]; trivial
  · rw [if_neg hv8]
    constructor
    · constructor
      · exact NotOccur.var 0 hv0
      · apply NotOccur.add
        · apply NotOccur.mul
          · exact NotOccur.var 8 hv8
```

```
        · apply NotOccur.succ
          apply NotOccur.mul
          · apply NotOccur.succ
            exact NotOccur.var 2 hv2
          · exact NotOccur.var 1 hv1
        · exact NotOccur.var 3 hv3
    · by_cases hv9 : v = 9
      · rw [if_pos hv9]; trivial
      · rw [if_neg hv9]
        constructor
        · apply NotOccur.succ
          apply NotOccur.mul
          · apply NotOccur.succ
            exact NotOccur.var 2 hv2
          · exact NotOccur.var 1 hv1
        · apply NotOccur.add
          · exact NotOccur.var 3 hv3
          · apply NotOccur.succ
            exact NotOccur.var 9 hv9


lemma NotFreeForm_PairForm_sh {s : Nat} (v : Nat)
    (hv0 : v ≠ 0) (hv1 : v ≠ 1) (hv2 : v ≠ 2) :
    NotFreeForm v (PairForm_sh (s := s)) := by
  dsimp [PairForm_sh, or_form, and_form, lt_form, exists_form, ArithForm.exists_form,
NotFreeForm]
  refine ⟨?_, ?_⟩
  · constructor
    · by_cases hv10 : v = 10
```

```
    · rw [if_pos hv10]; trivial
    · rw [if_neg hv10]
      constructor
      · exact NotOccur.var 1 hv1
      · apply NotOccur.add
        · exact NotOccur.var 0 hv0
        · apply NotOccur.succ
          exact NotOccur.var 10 hv10
  · constructor
    · exact NotOccur.var 2 hv2
    · apply NotOccur.add
      · apply NotOccur.mul
        · exact NotOccur.var 1 hv1
        · exact NotOccur.var 1 hv1
      · exact NotOccur.var 0 hv0
· constructor
  · by_cases hv10 : v = 10
    · rw [if_pos hv10]; trivial
    · rw [if_neg hv10]
      constructor
      · exact NotOccur.var 1 hv1
      · apply NotOccur.add
        · exact NotOccur.var 0 hv0
        · apply NotOccur.succ
          exact NotOccur.var 10 hv10
  · constructor
    · exact NotOccur.var 2 hv2
    · apply NotOccur.add
```

```
      · apply NotOccur.add
        · apply NotOccur.mul
          · exact NotOccur.var 0 hv0
          · exact NotOccur.var 0 hv0
        · exact NotOccur.var 0 hv0
      · exact NotOccur.var 1 hv1


lemma NotFreeForm_IsSuccCode {s : Nat} (v : Nat)
    (hv5 : v ≠ 5) (hv6 : v ≠ 6) :
    NotFreeForm v (IsSuccCode (s := s)) := by
  dsimp [IsSuccCode, exists_form, and_form, NotFreeForm]
  by_cases hv7 : v = 7
  · rw [if_pos hv7]; trivial
  · rw [if_neg hv7]
    constructor
    · dsimp [substForm, substTerm, PairForm_sh, or_form, and_form, lt_form,
exists_form, ArithForm.exists_form, NotFreeForm]
      refine ⟨?_, ?_⟩
      · constructor
        · by_cases hv10 : v = 10
          · rw [if_pos hv10]; trivial
          · rw [if_neg hv10]
            constructor
            · exact NotOccur.var 5 hv5
            · apply NotOccur.add
              · exact NotOccur_of_isClosed (natToTerm 4) (isClosed_natToTerm 4) v
              · apply NotOccur.succ
                exact NotOccur.var 10 hv10
```

```
  · constructor
    · exact NotOccur.var 7 hv7
    · apply NotOccur.add
      · apply NotOccur.mul
        · exact NotOccur.var 5 hv5
        · exact NotOccur.var 5 hv5
      · exact NotOccur_of_isClosed (natToTerm 4) (isClosed_natToTerm 4) v
· constructor
  · by_cases hv10 : v = 10
    · rw [if_pos hv10]; trivial
    · rw [if_neg hv10]
      constructor
      · exact NotOccur.var 5 hv5
      · apply NotOccur.add
        · exact NotOccur_of_isClosed (natToTerm 4) (isClosed_natToTerm 4) v
        · apply NotOccur.succ
          exact NotOccur.var 10 hv10
  · constructor
    · exact NotOccur.var 7 hv7
    · apply NotOccur.add
      · apply NotOccur.add
        · apply NotOccur.mul
          · exact NotOccur_of_isClosed (natToTerm 4) (isClosed_natToTerm 4) v
          · exact NotOccur_of_isClosed (natToTerm 4) (isClosed_natToTerm 4) v
        · exact NotOccur_of_isClosed (natToTerm 4) (isClosed_natToTerm 4) v
      · exact NotOccur.var 5 hv5
· constructor
  · exact NotOccur.var 6 hv6
```

· apply NotOccur.succ

exact NotOccur.var 7 hv7

Lemas auxiliares de frescura en las subfórmulas principales de `NumForm`

Se demuestran cuatro lemas que establecen condiciones de no ocurrencia para las subfórmulas que componen `NumForm`. Estos lemas garantizan que las variables ligadas de `NumForm` no interfieren con las variables libres ni con otras variables ligadas, lo cual es indispensable para la manipulación correcta de sustituciones en las demostraciones de representabilidad.

Lema `NotFreeForm_C1`: Frescura en el caso base de la secuencia

Establece que cualquier variable v distinta de 12 y 13 no ocurre libre en la subfórmula que expresa el caso base de la construcción de numerales:

$$ \text{BetaForm_sh}(\overline{a}, \overline{b}, \overline{0}, \overline{\text{zeroCode}}) $$

donde $\overline{a}$ y $\overline{b}$ están representados por las variables ligadas v_{12} y v_{13} (que serán cuantificadas existencialmente en `NumForm`). La demostración verifica que todos los componentes de `BetaForm_sh` que no son v_{12} o v_{13} están ligados internamente (variables 8 y 9) o son numerales cerrados ($\overline{0}$, $\overline{\text{zeroCode}}$), por lo que ninguna variable distinta de 12 y 13 puede aparecer libre.

Lema `NotFreeForm_C2`: Frescura en el elemento final de la secuencia

Establece que cualquier variable v distinta de 0, 1, 12 y 13 no ocurre libre en la subfórmula que expresa la condición del elemento final:

$$ \text{BetaForm_sh}(\overline{a}, \overline{b}, v_0, v_1) $$

Aquí v_0 y v_1 son las variables libres de `NumForm` (que representan w y z respectivamente), y $\overline{a}$, $\overline{b}$ son nuevamente v_{12} y v_{13}. Las variables 0 y 1 sí aparecen libres (son los argumentos de `BetaForm_sh`), pero todas las demás variables con índices distintos están ligadas o son cerradas.

Lema `NotFreeForm_C3`: Frescura en la condición de orden $k < w$

Establece que cualquier variable v distinta de 0 y 4 no ocurre libre en la subfórmula:

$$ v_4 < v_0 \quad \text{(con variable testigo 11)} $$

donde v_0 es la variable libre de `NumForm` y v_4 es la variable de inducción k (que está ligada por el cuantificador universal en `NumForm`). La variable testigo 11 está ligada por el existencial implícito en `lt_form`. La demostración verifica que ambas variables (0 y 4) son las únicas que pueden aparecer libres.

Lema `NotFreeForm_C4`: Frescura en la condición de paso inductivo

Establece que cualquier variable v distinta de 4, 5, 12 y 13 no ocurre libre en la subfórmula que expresa $\beta(a, b, k) = v$:

$$ \text{BetaForm_sh}(\overline{a}, \overline{b}, v_4, v_5) $$

donde v_4 es la variable de inducción k y v_5 es el valor intermedio v (ambas ligadas por cuantificadores existenciales en `NumForm`). Las variables 12 y 13 representan $\overline{a}$ y $\overline{b}$. Las variables internas de `BetaForm_sh` (8 y 9) están ligadas y no causan conflicto.

Significado para la estructura de `NumForm`

Estos lemas de frescura son técnicamente indispensables para la demostración de que `NumForm` satisface las condiciones de `IsRepresentable`. Específicamente, garantizan que:

1. **Variables libres correctas:** `NumForm` tiene exactamente v_0 y v_1 como variables libres. Todas las demás variables que aparecen en las subfórmulas están ligadas por cuantificadores (v_4 por $\forall$, v_5 y v_6 por $\exists$, v_{12} y v_{13} por los existenciales externos) o son variables internas de subfórmulas (v_7 a v_{11}).

2. **Sustituciones seguras:** Cuando se instancia `NumForm` con numerales concretos para v_0 y v_1, las sustituciones no causan capturas de variables ligadas, porque las variables de sustitución (0 y 1) son distintas de todas las variables ligadas en la fórmula.

3. **Clausura bajo cuantificación:** Las subfórmulas `C1` a `C4` están diseñadas para que las variables v_{12} y v_{13} actúen como parámetros que serán ligados por los cuantificadores existenciales más externos de `NumForm`. Esto permite que la

fórmula exprese "existe una secuencia codificada por a y b" sin que estos parámetros aparezcan libres en la fórmula completa.

La verificación explícita de estas condiciones de frescura, aunque tediosa, es una garantía formal de que la construcción sintáctica de `NumForm` es correcta y no contiene errores de ligadura. En sistemas como Lean, donde la distinción entre variables libres y ligadas se maneja a nivel del metalenguaje mediante nombres, estas demostraciones son necesarias para establecer propiedades que en una presentación con índices de De Bruijn serían inmediatas por construcción.

```
--
============================================================================
-- LEMAS AUXILIARES DE FRESCURA EN SUBFÓRMULAS PRINCIPALES
--
============================================================================

lemma NotFreeForm_C1 {s : Nat} (v : Nat) (hv12 : v ≠ 12) (hv13 : v ≠ 13) :
   NotFreeForm v (substForm 3 (natToTerm (s := s) zeroCode)
    (substForm 2 (natToTerm (s := s) 0)
     (substForm 1 (ArithTerm.var (s := s) 13)
      (substForm 0 (ArithTerm.var (s := s) 12) (BetaForm_sh (s := s)))))) := by
 dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form,
NotFreeForm, substForm, substTerm, natToTerm]
 by_cases hv8 : v = 8
```

```
  · rw [if_pos hv8]
    trivial
  · rw [if_neg hv8]
    refine ⟨⟨NotOccur.var 12 hv12, ?_⟩, ?_⟩
    · apply NotOccur.add
      · apply NotOccur.mul
        · exact NotOccur.var 8 hv8
        · apply NotOccur.succ
          apply NotOccur.mul
          · apply NotOccur.succ
            exact NotOccur_of_isClosed (natToTerm (s := s) 0) (isClosed_natToTerm 0) v
          · exact NotOccur.var 13 hv13
      · exact NotOccur_of_isClosed (natToTerm (s := s) zeroCode) (isClosed_natToTerm
zeroCode) v
    · by_cases hv9 : v = 9
      · rw [if_pos hv9]
        trivial
      · rw [if_neg hv9]
        refine ⟨?_, ?_⟩
        · apply NotOccur.succ
          apply NotOccur.mul
          · apply NotOccur.succ
            exact NotOccur_of_isClosed (natToTerm (s := s) 0) (isClosed_natToTerm 0) v
          · exact NotOccur.var 13 hv13
        · apply NotOccur.add
          · exact NotOccur_of_isClosed (natToTerm (s := s) zeroCode)
(isClosed_natToTerm zeroCode) v
          · apply NotOccur.succ
            exact NotOccur.var 9 hv9
```

```
lemma NotFreeForm_C2 {s : Nat} (v : Nat) (hv0 : v ≠ 0) (hv1 : v ≠ 1) (hv12 : v ≠ 12)
(hv13 : v ≠ 13) :

  NotFreeForm v (substForm 3 (ArithTerm.var (s := s) 1)

   (substForm 2 (ArithTerm.var (s := s) 0)

    (substForm 1 (ArithTerm.var (s := s) 13)

     (substForm 0 (ArithTerm.var (s := s) 12) (BetaForm_sh (s := s)))))) := by

 dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form,
NotFreeForm, substForm, substTerm]

 by_cases hv8 : v = 8

 · rw [if_pos hv8]

  trivial

 · rw [if_neg hv8]

  refine ⟨⟨NotOccur.var 12 hv12, ?_⟩, ?_⟩

  · apply NotOccur.add

   · apply NotOccur.mul

    · exact NotOccur.var 8 hv8

    · apply NotOccur.succ

     apply NotOccur.mul

     · apply NotOccur.succ

      exact NotOccur.var 0 hv0

     · exact NotOccur.var 13 hv13

   · exact NotOccur.var 1 hv1

  · by_cases hv9 : v = 9

   · rw [if_pos hv9]

    trivial

   · rw [if_neg hv9]
```

```
    refine ⟨?_, ?_⟩
    · apply NotOccur.succ
      apply NotOccur.mul
      · apply NotOccur.succ
        exact NotOccur.var 0 hv0
      · exact NotOccur.var 13 hv13
    · apply NotOccur.add
      · exact NotOccur.var 1 hv1
      · apply NotOccur.succ
        exact NotOccur.var 9 hv9
```

```
lemma NotFreeForm_C3 {s : Nat} (v : Nat) (hv0 : v ≠ 0) (hv4 : v ≠ 4) :
    NotFreeForm v (lt_form (ArithTerm.var (s := s) 4) (ArithTerm.var (s := s) 0) 11 :
ArithForm s) := by
  dsimp [lt_form, exists_form, ArithForm.exists_form, NotFreeForm]
  by_cases hv11 : v = 11
  · rw [if_pos hv11]
    trivial
  · rw [if_neg hv11]
    refine ⟨NotOccur.var 0 hv0, ?_⟩
    apply NotOccur.add
    · exact NotOccur.var 4 hv4
    · apply NotOccur.succ
      exact NotOccur.var 11 hv11
```

```
lemma NotFreeForm_C4 {s : Nat} (v : Nat) (hv4 : v ≠ 4) (hv5 : v ≠ 5) (_hv8 : v ≠ 8) (_hv8
: v ≠ 9) (hv12 : v ≠ 12) (hv13 : v ≠ 13) :

  NotFreeForm v (substForm 3 (ArithTerm.var (s := s) 5)

   (substForm 2 (ArithTerm.var (s := s) 4)

    (substForm 1 (ArithTerm.var (s := s) 13)

     (substForm 0 (ArithTerm.var (s := s) 12) (BetaForm_sh (s := s)))))) := by

 dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form,
NotFreeForm, substForm, substTerm]

 by_cases hv8' : v = 8

 · rw [if_pos hv8']

  trivial

 · rw [if_neg hv8']

  refine ⟨⟨NotOccur.var 12 hv12, ?_⟩, ?_⟩

  · apply NotOccur.add

   · apply NotOccur.mul

    · exact NotOccur.var 8 hv8'

    · apply NotOccur.succ

     apply NotOccur.mul

     · apply NotOccur.succ

      exact NotOccur.var 4 hv4

     · exact NotOccur.var 13 hv13

   · exact NotOccur.var 5 hv5

  · by_cases hv9' : v = 9

   · rw [if_pos hv9']

    trivial

   · rw [if_neg hv9']
```

```
    refine ⟨?_, ?_⟩
    · apply NotOccur.succ
      apply NotOccur.mul
      · apply NotOccur.succ
        exact NotOccur.var 4 hv4
      · exact NotOccur.var 13 hv13
    · apply NotOccur.add
      · exact NotOccur.var 5 hv5
      · apply NotOccur.succ
        exact NotOccur.var 9 hv9'
```

Lema `NotFreeForm_C5`: Frescura en la condición de paso inductivo (segunda parte)

Establece que cualquier variable v distinta de 4, 6, 12 y 13 no ocurre libre en la subfórmula que expresa $\beta(a, b, k+1) = v'$:

$$ \text{BetaForm_sh}(\overline{a}, \overline{b}, S(v_4), v_6) $$

donde v_4 es la variable de inducción k y v_6 es el valor intermedio v'. La presencia de $S(v_4)$ en la posición del argumento I de `BetaForm_sh` introduce una pequeña complicación adicional respecto a `C4`, pero la estructura de la demostración es análoga: se verifica que todas las variables con índices distintos de

4, 6, 12, 13 están ligadas internamente (variables 8 y 9 de `BetaForm_sh`) o son términos que no contienen variables libres adicionales. El sucesor aplicado a v_4 no introduce nuevas variables libres; simplemente envuelve la variable existente.

Lema `NotFreeForm_NumForm_of_ge_12`: Condensación de frescura para variables de índice alto

Este lema constituye la verificación unificada de que cualquier variable con índice $v \geq 12$ no ocurre libre en `NumForm`. Es un resultado técnico que consolida todos los lemas de frescura anteriores en una única declaración.

Estructura de la demostración

La prueba expande completamente la definición de `NumForm` y analiza cada subfórmula:

1. **Cuantificadores existenciales externos (v_{12} y v_{13}):** Si $v = 12$ o $v = 13$, la variable está ligada por los cuantificadores $\exists v_{12}$ y $\exists v_{13}$ respectivamente, por lo que no es libre.

2. **Cuantificador universal (v_4):** La variable v_4 está ligada por el $\forall k$, por lo que no es libre. Esto se refleja en la condición `if_neg hv4` (pues $v \geq 12 \implies v \neq 4$).

3. **Cuantificadores existenciales internos (v_5, v_6):** Las variables v_5 y v_6 están ligadas por los $\exists v, v'$, por lo que no son libres.

4. **Variables internas de subfórmulas (v_7 a v_{11}):** Las variables 7 (de `IsSuccCode`), 8 y 9 (de `BetaForm_sh`), 10 (de `PairForm_sh`), y 11 (de `lt_form`) están todas ligadas en sus respectivas subfórmulas.

5. **Subfórmulas atómicas:** Para las cinco subfórmulas componentes (`C1` a `C5` e `IsSuccCode`), se invocan los lemas de frescura correspondientes, todos los cuales requieren que v sea distinto de las variables libres de cada subfórmula. Dado que $v \geq 12$, automáticamente v es distinto de $0, 1, 2, 3, 4, 5, 6$, satisfaciendo todas las hipótesis de dichos lemas.

Significado

Este lema es la pieza final que garantiza que `NumForm` tiene exactamente dos variables libres: v_0 y v_1. Cualquier variable con índice $v \geq 2$ (y en particular $v \geq 12$) está ligada en algún cuantificador de la fórmula o en una de sus subfórmulas. Esta propiedad es esencial para que `NumForm` pueda funcionar como fórmula representadora en el sentido de `IsRepresentable`, donde se exige que solo v_0 y v_1 sean libres. La demostración, aunque extensa, sigue un patrón sistemático: para cada variable ligada en la fórmula, se verifica que la condición de `NotFreeForm` se satisface porque el constructor correspondiente (cuantificador existencial o universal) establece la condición de ligadura.

```
lemma NotFreeForm_C5 {s : Nat} (v : Nat) (hv4 : v ≠ 4) (hv6 : v ≠ 6) (_hv8 : v ≠ 8) (_hv9 : v ≠ 9) (hv12 : v ≠ 12) (hv13 : v ≠ 13) :

  NotFreeForm v (substForm 3 (ArithTerm.var (s := s) 6)

   (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))

    (substForm 1 (ArithTerm.var (s := s) 13)

     (substForm 0 (ArithTerm.var (s := s) 12) (BetaForm_sh (s := s)))))) := by

 dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, NotFreeForm, substForm, substTerm]

 by_cases hv8' : v = 8
```

· rw [if_pos hv8']
  trivial
· rw [if_neg hv8']
  refine ⟨⟨NotOccur.var 12 hv12, ?_⟩, ?_⟩
  · apply NotOccur.add
    · apply NotOccur.mul
      · exact NotOccur.var 8 hv8'
      · apply NotOccur.succ
        apply NotOccur.mul
        · apply NotOccur.succ
          apply NotOccur.succ
          exact NotOccur.var 4 hv4
        · exact NotOccur.var 13 hv13
    · exact NotOccur.var 6 hv6
  · by_cases hv9' : v = 9
    · rw [if_pos hv9']
      trivial
    · rw [if_neg hv9']
      refine ⟨?_, ?_⟩
      · apply NotOccur.succ
        apply NotOccur.mul
        · apply NotOccur.succ
          apply NotOccur.succ
          exact NotOccur.var 4 hv4
        · exact NotOccur.var 13 hv13
      · apply NotOccur.add
        · exact NotOccur.var 6 hv6
        · apply NotOccur.succ

```
      exact NotOccur.var 9 hv9'



lemma NotFreeForm_NumForm_of_ge_12 {s : Nat} (v : Nat) (hv12 : v ≥ 12) :
    NotFreeForm v (NumForm (s := s)) := by
  dsimp [NumForm, BetaForm_sh, PairForm_sh, lt_form, exists_form,
ArithForm.exists_form, and_form, or_form, NotFreeForm]
  have hv2 : v ≠ 2 := by omega
  have hv3 : v ≠ 3 := by omega
  have hv4 : v ≠ 4 := by omega
  have hv5 : v ≠ 5 := by omega
  have hv6 : v ≠ 6 := by omega
  have hv7 : v ≠ 7 := by omega
  have hv8 : v ≠ 8 := by omega
  have hv9 : v ≠ 9 := by omega
  have hv10 : v ≠ 10 := by omega
  have hv11 : v ≠ 11 := by omega
  by_cases hv12' : v = 12
  · rw [if_pos hv12']; trivial
  · rw [if_neg hv12']
    by_cases hv13' : v = 13
    · rw [if_pos hv13']; trivial
    · rw [if_neg hv13']
      rw [if_neg hv4, if_neg hv11, if_neg hv5, if_neg hv6]
      refine ⟨?_, ?_, ?_, ?_, ?_, ?_⟩
      · exact NotFreeForm_C1 v (by omega) (by omega)
      · exact NotFreeForm_C2 v (by omega) (by omega) (by omega) (by omega)
```

· refine ⟨NotOccur.var 0 (by omega), ?_⟩

exact NotOccur.add (ArithTerm.var 4) ((ArithTerm.var 11).succ) (NotOccur.var 4 hv4) (NotOccur.succ (ArithTerm.var 11) (NotOccur.var 11 hv11))

· exact NotFreeForm_C4 v hv4 hv5 hv8 hv9 (by omega) (by omega)

· exact NotFreeForm_C5 v hv4 hv6 hv8 hv9 (by omega) (by omega)

· exact NotFreeForm_IsSuccCode v hv5 hv6

Teorema principal `NumForm_free_vars`: Caracterización completa de variables libres de `NumForm`

Se demuestra el teorema que establece de manera exhaustiva qué variables pueden aparecer libres en `NumForm`. El resultado es que **solo las variables v_0 y v_1 son libres**; cualquier otra variable v que no sea 0 ni 1 satisface `NotFreeForm v NumForm`, bajo las hipótesis adicionales de que v es distinta de las variables ligadas internas.

Formalmente, para cualquier variable v tal que:

$$ v \neq 0,\ v \neq 1,\ v \neq 4,\ v \neq 5,\ v \neq 6,\ v \neq 7,\ v \neq 8,\ v \neq 9,\ v \neq 10,\ v \neq 11,\ v \neq 12,\ v \neq 13 $$

se tiene $\text{NotFreeForm } v\ \text{NumForm}$.

Estructura de la demostración

La prueba expande la definición de `NumForm` y procede por análisis exhaustivo de casos, verificando que cada variable ligada en la fórmula está efectivamente ligada y no es libre:

1. Cuantificadores existenciales externos: Si $v = 12$ o $v = 13$, estas variables están ligadas por los cuantificadores $\exists v_{12}$ y $\exists v_{13}$ que envuelven toda la fórmula, por lo que `NotFreeForm` se satisface trivialmente (la condición `if_pos` en la definición de `NotFreeForm` para `exists_form`).

2. Cuantificador universal (v_4): Si $v = 4$, está ligada por el $\forall k$ que cuantifica sobre la variable de inducción. La condición `if_pos` correspondiente a `forall_form` captura este caso.

3. Variables ligadas en subfórmulas:

- v_{11}: ligada en `lt_form` (variable testigo del existencial interno)

- v_5 y v_6: ligadas por los cuantificadores existenciales $\exists v, v'$

- Las variables 7, 8, 9, 10 están ligadas dentro de `IsSuccCode`, `BetaForm_sh` y `PairForm_sh` respectivamente

4. Variables restantes: Para cualquier v que no coincida con ninguna de las variables ligadas, se invocan los lemas de frescura `C1` a `C5` e `IsSuccCode`, los cuales garantizan que v no aparece libre en ninguna de las seis subfórmulas que componen la conjunción dentro de `NumForm`.

Significado del teorema

`NumForm_free_vars` es el resultado que certifica que `NumForm` es una fórmula bien comportada desde el punto de vista de las variables libres, cumpliendo exactamente el requisito estructural de `IsRepresentable`: tener solo v_0 y v_1 como variables libres.

Este teorema es el resultado de una verificación meticulosa de la estructura de ligaduras de una fórmula considerablemente compleja. La lista de hipótesis (doce desigualdades) refleja la cantidad de variables ligadas que contiene `NumForm`: dos para los parámetros de la función beta (v_{12}, v_{13}), una para la inducción (v_4), dos para los valores intermedios (v_5, v_6), una para `IsSuccCode` (v_7), dos para `BetaForm_sh` (v_8, v_9), una para `PairForm_sh` (v_{10}), y una para `lt_form` (v_{11}). En total, diez variables ligadas además de las dos libres (v_0, v_1), lo que da cuenta de la complejidad de la construcción.

La demostración, aunque extensa, sigue un patrón mecánico: para cada variable ligada, se verifica que el constructor sintáctico correspondiente (cuantificador existencial, cuantificador universal, o variable testigo en `lt_form`) establece la condición de `NotFreeForm`. Esta verificación sistemática es esencial para garantizar que las sustituciones que se realizarán sobre v_0 y v_1 cuando se use `NumForm` como fórmula representadora no causarán capturas de variables ligadas.

--
==

-- TEOREMA PRINCIPAL: NumForm_free_vars

--
==

```
theorem NumForm_free_vars {s : Nat} (v : Nat) (hv0 : v ≠ 0) (hv1 : v ≠ 1) (hv12 : v ≠ 12) (hv13 : v ≠ 13) (hv4 : v ≠ 4) (hv5 : v ≠ 5) (hv6 : v ≠ 6) (_hv7 : v ≠ 7) (hv8 : v ≠ 8) (hv9 : v ≠ 9) (_hv10 : v ≠ 10) (hv11 : v ≠ 11) :
  NotFreeForm v (NumForm (s := s)) := by
 dsimp [NumForm, exists_form, and_form, lt_form, ArithForm.exists_form, NotFreeForm, substForm, substTerm]
 by_cases hv12' : v = 12
 · rw [if_pos hv12']; trivial
 · rw [if_neg hv12']
  by_cases hv13' : v = 13
  · rw [if_pos hv13']; trivial
  · rw [if_neg hv13']
   by_cases hv4' : v = 4
   · rw [if_pos hv4']; trivial
   · rw [if_neg hv4']
    by_cases hv11' : v = 11
    · rw [if_pos hv11']; trivial
    · rw [if_neg hv11']
     by_cases hv5' : v = 5
     · rw [if_pos hv5']; trivial
     · rw [if_neg hv5']
      by_cases hv6' : v = 6
      · rw [if_pos hv6']; trivial
      · rw [if_neg hv6']
       refine ⟨?_, ?_, ?_, ?_, ?_, ?_⟩
       · exact NotFreeForm_C1 v hv12 hv13
```

· exact NotFreeForm_C2 v hv0 hv1 hv12 hv13

· refine ⟨NotOccur.var 0 hv0, ?_⟩

exact NotOccur.add (ArithTerm.var 4) ((ArithTerm.var 11).succ) (NotOccur.var 4 hv4') (NotOccur.succ (ArithTerm.var 11) (NotOccur.var 11 hv11'))

· exact NotFreeForm_C4 v hv4' hv5' hv8 hv9 hv12 hv13

· exact NotFreeForm_C5 v hv4' hv6' hv8 hv9 hv12 hv13

· exact NotFreeForm_IsSuccCode v hv5 hv6

Lemas auxiliares del metalenguaje para la función beta de Gödel

Se demuestran tres lemas en el metalenguaje (sobre $\mathbb{N}$) que establecen propiedades aritméticas fundamentales necesarias para la construcción de la función beta. Estos resultados no involucran al sistema formal SPA directamente, sino que establecen hechos sobre números naturales que luego serán utilizados para guiar la representabilidad interna.

Lema `le_maxSeq`: Acotación de una función por su máximo acumulado

Define la función `maxSeq f n` como el máximo de los valores $f(0), f(1), \ldots, f(n)$, y demuestra que cualquier valor $f(i)$ con $i \leq n$ está acotado por este máximo:

$$ \forall f : \mathbb{N} \to \mathbb{N},\ \forall n, i \in \mathbb{N},\ i \leq n \implies f(i) \leq \text{maxSeq}(f, n) $$

Estructura de la demostración

La prueba procede por inducción sobre n:

- **Caso base ($n = 0$):** Si $i \leq 0$, entonces $i = 0$ y $f(0) \leq \text{maxSeq}(f, 0)$ por definición.

- **Paso inductivo ($n' \to n'+1$):** Se distingue si $i = n'+1$ o $i \leq n'$:

 - Si $i = n'+1$, entonces $f(i) \leq \max(f(n'+1), \text{maxSeq}(f, n')) = \text{maxSeq}(f, n'+1)$.

 - Si $i \leq n'$, por hipótesis de inducción $f(i) \leq \text{maxSeq}(f, n')$, y este último está acotado por el máximo con $f(n'+1)$.

Significado

Este lema es una herramienta técnica para construir la cota b en la función beta de Gödel. En la codificación de secuencias, se necesita un número b que sea divisible por todos los números de 1 a n, y el máximo de la secuencia proporciona una cota para los valores a codificar.

Lema `coprime_moduli_beta`: Coprimalidad de los módulos de la función beta

Establece que los módulos $1 + (i+1) \cdot B$ y $1 + (j+1) \cdot B$ son coprimos para $i \neq j$, siempre que B sea divisible por todos los números de 1 a n:

$$ \forall B \geq 1,\ \forall n \in \mathbb{N},\ (\forall k \in [1, n],\ k \mid B) \implies $$

$$ \forall i, j \leq n,\ i < j \implies \text{Coprime}(1 + (i+1) \cdot B,\ 1 + (j+1) \cdot B) $$

Estructura de la demostración

La prueba utiliza la caracterización de la coprimalidad mediante divisores comunes:

1. Sea p un primo que divide a ambos módulos: $p \mid 1 + (i+1)B$ y $p \mid 1 + (j+1)B$. Existen k_i, k_j tales que $p \cdot k_i = 1 + (i+1)B$ y $p \cdot k_j = 1 + (j+1)B$.

2. La diferencia de ambas igualdades da $(j - i) \cdot B = p \cdot (k_j - k_i)$.

3. Por tanto, $p \mid (j - i) \cdot B$. Como p es primo, $p \mid (j - i)$ o $p \mid B$.

4. En cualquier caso, $p \mid B$:

  - Si $p \mid (j - i)$, como $1 \leq j - i \leq n$, la hipótesis sobre B implica $(j - i) \mid B$, de donde $p \mid B$.

  - Si $p \mid B$, es inmediato.

5. Sea $B = p \cdot k_B$. Sustituyendo en la primera igualdad: $p \cdot k_i = 1 + (i+1) \cdot p \cdot k_B = 1 + p \cdot ((i+1) \cdot k_B)$.

6. Reordenando: $p \cdot (k_i - (i+1) \cdot k_B) = 1$, lo que implica $p \mid 1$.

7. Por tanto, $p = 1$. Pero un primo es al menos 2, contradicción. Luego no existe tal primo, y los módulos son coprimos.

Lema `pairwise_coprime_range`: Coprimalidad por pares sobre un rango

Establece que si una función s produce valores coprimos dos a dos para índices en un rango, entonces la lista de valores sobre ese rango satisface la propiedad `Pairwise` de coprimalidad:

$$ \forall s : \mathbb{N} \to \mathbb{N},\ \forall n \in \mathbb{N}, $$

$$ (\forall i, j \leq n,\ i < j \implies \text{Coprime}(s(i), s(j))) \implies \text{Pairwise Coprime } [s(0), s(1), \ldots, s(n)] $$

Estructura de la demostración

La prueba procede por inducción sobre n:

- **Caso base ($n = 0$):** La lista tiene un solo elemento, y `Pairwise` se satisface trivialmente.

- **Paso inductivo ($n' \to n'+1$):** Se utiliza la propiedad de que `List.pairwise_append` descompone la verificación en tres condiciones:

  1. La lista sin el último elemento satisface `Pairwise` (hipótesis de inducción).

  2. La lista con solo el último elemento satisface `Pairwise` (trivial).

  3. Cada elemento anterior es coprimo con el nuevo elemento (se sigue de la hipótesis general aplicada a i y $n'+1$).

Significado para la función beta de Gödel

Estos tres lemas trabajan conjuntamente para establecer el fundamento aritmético del Teorema Chino del Resto, que es la base de la función beta de Gödel:

1. **`le_maxSeq`** permite acotar los valores de una secuencia finita.

2. **`coprime_moduli_beta`** establece que eligiendo B como el factorial de n (o cualquier número divisible por todos los números hasta n), los módulos $1 + (i+1)B$ son coprimos dos a dos.

3. **`pairwise_coprime_range`** reformula esta propiedad en términos de listas, preparando el terreno para aplicar el Teorema Chino del Resto, que garantiza la existencia de un número A tal que $A \equiv a_i \pmod{1 + (i+1)B}$ para una secuencia dada $(a_0, \ldots, a_n)$.

Estos resultados del metalenguaje son la contraparte semántica de la representabilidad de `BetaForm`: garantizan que la función beta está bien definida y tiene las propiedades necesarias para codificar secuencias, lo cual es el fundamento último de la aritmetización de la sintaxis. La demostración en el sistema formal SPA de que `BetaForm` representa correctamente esta función se apoya implícitamente en estas propiedades, aunque la prueba interna utiliza métodos sintácticos (inducción, maximalidad, Henkin) en lugar de apelar directamente al Teorema Chino del Resto.

--
==

-- SUB-ETAPA 1: TEOREMA DE LA FUNCIÓN BETA DE GÖDEL EN NAT (METALENGUAJE)

--
==

```
-- Lema auxiliar de acotación recursiva para funciones en Nat
def maxSeq (f : Nat → Nat) : Nat → Nat
 | 0 => f 0
 | k + 1 => Nat.max (f (k + 1)) (maxSeq f k)

lemma le_maxSeq (f : Nat → Nat) (n : Nat) (i : Nat) (hi : i ≤ n) : f i ≤ maxSeq f n := by
 induction n with
 | zero =>
  have h_i : i = 0 := by omega
  subst h_i
  rfl
 | succ n' ih =>
  by_cases h_eq : i = n' + 1
  · subst h_eq
   dsimp [maxSeq]
   exact Nat.le_max_left _ _
  · have hi' : i ≤ n' := by omega
   have h_le := ih hi'
   dsimp [maxSeq]
   exact Nat.le_trans h_le (Nat.le_max_right _ _)

lemma coprime_moduli_beta (B : Nat) (_hB : B ≥ 1) (n : Nat) (hB_div : ∀ k, 1 ≤ k → k ≤ n
→ k ∣ B)
```

```
  (i j : Nat) (_hi : i ≤ n) (hj : j ≤ n) (hij : i < j) :
  Nat.Coprime (1 + (i + 1) * B) (1 + (j + 1) * B) := by
apply Nat.coprime_of_dvd
intro p hp hp_i hp_j
obtain ⟨ki, hki⟩ := hp_i
obtain ⟨kj, hkj⟩ := hp_j
have h_diff_eq : (j - i) * B = p * (kj - ki) := by
  have h_add : (j + 1) * B = (i + 1) * B + (j - i) * B := by
    rw [← Nat.add_mul]
    congr 1
    omega
  have h_sub1 : 1 + (j + 1) * B - (1 + (i + 1) * B) = (j - i) * B := by omega
  have h_sub2 : 1 + (j + 1) * B - (1 + (i + 1) * B) = p * (kj - ki) := by
    rw [hkj, hki]
    rw [← Nat.mul_sub_left_distrib]
  rw [← h_sub1, h_sub2]
have h_diff : p ∣ (j - i) * B := ⟨kj - ki, h_diff_eq⟩
have hp_or := (Nat.Prime.dvd_mul hp).mp h_diff
have hp_B : p ∣ B := by
  rcases hp_or with hp_ji | hp_B
  · have h_k1 : 1 ≤ j - i := by omega
    have h_kn : j - i ≤ n := by omega
    have h_ji_dvd_B := hB_div (j - i) h_k1 h_kn
    exact Nat.dvd_trans hp_ji h_ji_dvd_B
  · exact hp_B
obtain ⟨kB, hkB⟩ := hp_B
have h_i1 : (i + 1) * B = p * ((i + 1) * kB) := by
  rw [hkB]
```

```
  ring
 rw [h_i1] at hki
 have h_sub_1 : 1 = p * ki - p * ((i + 1) * kB) := by omega
 have h_dist : p * ki - p * ((i + 1) * kB) = p * (ki - (i + 1) * kB) := by
  rw [← Nat.mul_sub_left_distrib]
 have h_one_eq : 1 = p * (ki - (i + 1) * kB) := by
  rw [← h_dist]
  exact h_sub_1
 have hp_dvd_1 : p ∣ 1 := ⟨ki - (i + 1) * kB, h_one_eq⟩
 have hp_eq_1 : p = 1 := Nat.dvd_one.mp hp_dvd_1
 have hp_ge_2 : p ≥ 2 := Nat.Prime.two_le hp
 omega




-- 3. Lema de soporte inductivo para List.Pairwise sobre List.range
lemma pairwise_coprime_range (s : Nat → Nat) (n : Nat)
   (h_cop : ∀ i j, i ≤ n → j ≤ n → i < j → Nat.Coprime (s i) (s j)) :
   List.Pairwise (Function.onFun Nat.Coprime s) (List.range (n + 1)) := by
 induction n with
 | zero =>
  rw [List.range_succ]
  exact List.Pairwise.nil.cons (by intro x hx; nomatch hx)
 | succ n' ih =>
  have h_ih : List.Pairwise (Function.onFun Nat.Coprime s) (List.range (n' + 1)) := by
    apply ih
```

```
  intro i j hi hj hij
  exact h_cop i j (by omega) (by omega) hij
rw [List.range_succ]
rw [List.pairwise_append]
refine ⟨h_ih, ?_, ?_⟩
· exact List.Pairwise.nil.cons (by intro x hx; nomatch hx)
· intro i hi j hj
  rw [List.mem_singleton] at hj
  subst hj
  rw [List.mem_range] at hi
  dsimp [Function.onFun]
  exact h_cop i (n' + 1) (by omega) (by omega) hi
```

Teorema principal `exists_beta_seq`: Existencia de parámetros para la función beta de Gödel

Se demuestra el teorema fundamental de existencia en el metalenguaje: para cualquier función finita f definida sobre $\{0, \ldots, n\}$, existen números naturales A y B tales que la función beta de Gödel recupera exactamente los valores de f:

$$ \forall f : \mathbb{N} \to \mathbb{N},\ \forall n \in \mathbb{N},\ \exists A, B \in \mathbb{N},\ \forall i \leq n,\ \text{godelBeta}(A, B, i) = f(i) $$

Estructura de la demostración

La prueba es una implementación constructiva del argumento clásico de Gödel basado en el Teorema Chino del Resto. Se desarrolla en cinco etapas:

1. Construcción de la cota N: Se define $N := \text{maxSeq}(f, n) + n + 2$, que es una cota superior estricta para todos los valores $f(i)$ (con $i \leq n$). La inclusión de $+n+2$ garantiza que $f(i) < N$ para todo $i \leq n$, y que N es suficientemente grande para los propósitos de la demostración.

2. Construcción de B: Se define $B := N!$ (el factorial de N). Esta elección asegura dos propiedades cruciales:

- $B \geq 1$ (trivialmente, pues $N \geq 1$).

- B es divisible por todo número k con $1 \leq k \leq n$, ya que $k \leq n \leq N$ y por tanto $k \mid N!$.

3. Construcción de los módulos: Se define la familia de módulos $s(i) := 1 + (i + 1) \cdot B$ para $i = 0, \ldots, n$. Por el lema `coprime_moduli_beta`, estos módulos son coprimos dos a dos, ya que B es divisible por todos los números de 1 a n.

4. Aplicación del Teorema Chino del Resto: Por la coprimalidad dos a dos de los módulos $s(i)$, el Teorema Chino del Resto (en su versión para listas, `Nat.chineseRemainderOfList`) garantiza la existencia de un número A tal que para cada $i \leq n$:

$$ A \equiv f(i) \pmod{s(i)} $$

es decir, $A \bmod s(i) = f(i) \bmod s(i)$.

5. Verificación de la función beta: Para cada $i \leq n$, se debe comprobar que $\text{godelBeta}(A, B, i) = f(i)$. Por definición:

$$ \text{godelBeta}(A, B, i) = A \bmod (1 + (i + 1) \cdot B) = A \bmod s(i) $$

De la congruencia del paso anterior, $A \bmod s(i) = f(i) \bmod s(i)$. Pero $f(i) \leq \text{maxSeq}(f, n) < N \leq B \leq (i+1) \cdot B < 1 + (i+1) \cdot B = s(i)$, por lo que $f(i) < s(i)$ y en consecuencia $f(i) \bmod s(i) = f(i)$. Por tanto, $\text{godelBeta}(A, B, i) = f(i)$.

Significado del teorema

`exists_beta_seq` es el resultado semántico que justifica toda la construcción de la función beta de Gödel. Establece que cualquier secuencia finita de números naturales puede ser codificada mediante dos números A y B, de modo que la función beta permite recuperar cada elemento de la secuencia a partir de su índice.

Este teorema es el análogo en el metalenguaje del resultado de representabilidad `beta_representable` en el sistema formal SPA. Mientras que `beta_representable` demuestra que la relación $R = \beta(A, B, I)$ es expresable y funcional dentro del sistema formal, `exists_beta_seq` demuestra que externamente la función beta tiene la propiedad de codificación universal de secuencias finitas.

La combinación de ambos resultados —el externo de existencia y el interno de representabilidad— es lo que permite a Gödel codificar afirmaciones sobre secuencias de fórmulas (como las demostraciones) dentro del sistema formal. En particular:

1. Dada una secuencia finita de códigos de fórmulas $\langle c_0, \ldots, c_n \rangle$, el teorema `exists_beta_seq` proporciona A, B tales que $\beta(A, B, i) = c_i$.

2. La fórmula `BetaForm` permite expresar internamente la relación $c_i = \beta(A, B, i)$.

3. `NumForm` extiende esta idea para expresar que un código corresponde a un numeral, construyendo inductivamente la secuencia de códigos desde `zero` hasta el numeral deseado.

Esta arquitectura —existencia externa más representabilidad interna— es el núcleo técnico de la aritmetización de la sintaxis y, en última instancia, de los teoremas de incompletitud.

```
-- 4. Teorema Principal de la Sub-etapa 1: Existencia de parámetros Beta de Gödel
theorem exists_beta_seq (f : Nat → Nat) (n : Nat) :
  ∃ A B : Nat, ∀ i ≤ n, godelBeta A B i = f i := by
 let N := maxSeq f n + n + 2
 let B := Nat.factorial N
 have hB_pos : B ≥ 1 := Nat.factorial_pos N
 have hB_div : ∀ k, 1 ≤ k → k ≤ n → k ∣ B := by
  intro k hk1 hkn
  have hkN : k ≤ N := by dsimp [N]; omega
  exact Nat.dvd_factorial hk1 hkN

 let s : Nat → Nat := fun i => 1 + (i + 1) * B
 let l : List Nat := List.range (n + 1)

 have h_pairwise : List.Pairwise (Function.onFun Nat.Coprime s) l := by
  dsimp [l]
```

```
  apply pairwise_coprime_range
  intro i j hi hj hij
  exact coprime_moduli_beta B hB_pos n hB_div i j hi hj hij

obtain ⟨A, h_modEq⟩ := Nat.chineseRemainderOfList f s l h_pairwise

use A, B
intro i hi
have h_mem : i ∈ l := by
  dsimp [l]
  exact List.mem_range.mpr (by omega)
have h_mod := h_modEq i h_mem
dsimp [godelBeta, s]
have h_fi_lt : f i < 1 + (i + 1) * B := by
  have h1 : f i ≤ maxSeq f n := le_maxSeq f n i hi
  have h2 : maxSeq f n < N := by dsimp [N]; omega
  have h3 : N ≤ B := Nat.self_le_factorial N
  have h4 : B ≤ (i + 1) * B := by
    have : (i + 1) * B = B + i * B := by rw [Nat.add_one_mul, Nat.add_comm]
    omega
  omega
rw [Nat.ModEq] at h_mod
have h_mod_self : f i % (1 + (i + 1) * B) = f i := Nat.mod_eq_of_lt h_fi_lt
rw [h_mod_self] at h_mod
exact h_mod
```

Sub-etapa 2.1: Equivalencia y representabilidad de `BetaForm_sh`

Se establece la equivalencia lógica entre `BetaForm` (que usa variables ligadas v_4 y v_5) y `BetaForm_sh` (que usa variables ligadas v_8 y v_9) en cualquier teoría Δ maximal consistente con propiedad de Henkin. Esta equivalencia es necesaria para evitar colisiones de variables cuando `BetaForm` se utiliza como subfórmula dentro de `NumForm`, donde las variables 4 y 5 ya están utilizadas para otros fines.

Definiciones auxiliares

`M0_term I B`: Término compuesto que representa el módulo $1 + (I+1) \cdot B$:

$$ M_0(I, B) := S(S(\overline{I}) \cdot \overline{B}) $$

`isClosed_M0` y `NotOccur_M0`: Propiedades de clausura y no ocurrencia para M_0, análogas a las de los numerales.

Lema `beta_sh_equivalence`: Equivalencia entre `BetaForm` y `BetaForm_sh`

Establece que para parámetros fijos A, B, I y un término cerrado R, las instancias de `BetaForm` y `BetaForm_sh` son equivalentes en Δ:

$$ \Delta \vdash \text{BetaForm}(\overline{A}, \overline{B}, \overline{I}, R) \iff \Delta \vdash \text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{I}, R) $$

Estructura de la demostración

1. Simplificación de ambas fórmulas: Tanto `BetaForm` como `BetaForm_sh`, tras instanciar las variables v_0, v_1, v_2, v_3 con $\overline{A}, \overline{B}, \overline{I}, R$, se reducen (mediante simplificación de sustituciones con `substTerm_fresh`) a formas existenciales que difieren solo en los índices de las variables ligadas:

- `BetaForm`: $\exists v_4\ (\overline{A} = v_4 \cdot M_0 + R \land \exists v_5\ (M_0 = R + S(v_5)))$

- `BetaForm_sh`: $\exists v_8\ (\overline{A} = v_8 \cdot M_0 + R \land \exists v_9\ (M_0 = R + S(v_9)))$

2. Equivalencia de las subfórmulas de orden: Se define $E_5 := (M_0 = R + S(v_5))$ y se aplica el lema `change_var_equivalence_strong` para establecer la equivalencia entre $\exists v_5\ E_5$ y $\exists v_9\ E_5[v_5 := v_9]$. Este lema requiere verificar que v_9 no ocurre libre en E_5 (lo cual es cierto porque M_0, R y v_5 son todos distintos de v_9) y que no hay conflictos de ligadura.

3. Equivalencia de las fórmulas completas: Se demuestran ambas direcciones de la equivalencia:

- **Dirección $\Rightarrow$:** Dado $\exists v_4\ (\overline{A} = v_4 \cdot M_0 + R \land \exists v_5\ E_5) \in \Delta$, por la propiedad de Henkin existe un testigo cerrado q para v_4. Simplificando la sustitución (usando que M_0, R, $\overline{A}$ son cerrados y que $\exists v_5\ E_5$ no contiene a v_4), se obtiene $\overline{A} = q \cdot M_0 + R \in \Delta$ y $\exists v_5\ E_5 \in \Delta$. Por la equivalencia de las subfórmulas de orden, $\exists v_9\ E_5[v_5 := v_9] \in \Delta$. Se reconstruye la conjunción y se introduce el existencial sobre v_8 con testigo q, obteniendo `BetaForm_sh` en Δ.

- **Dirección $\Leftarrow$:** Simétrica, usando el testigo q para v_8 y la equivalencia en dirección contraria.

Significado

Este lema de equivalencia es una pieza técnica esencial para la modularidad de la construcción. Permite utilizar `BetaForm_sh` (con variables ligadas 8 y 9) en lugar de `BetaForm` (con variables 4 y 5) dentro de `NumForm`, donde las variables 4 y 5 están reservadas para la variable de inducción k y el valor intermedio v. La demostración verifica que el cambio de variables ligadas (alpha-conversión) es lícito y no altera el significado de la fórmula en Δ, siempre que las nuevas variables no capturen variables libres de las subfórmulas.

La estructura de la prueba —reducción a formas explícitas mediante simplificación de sustituciones, uso de `change_var_equivalence_strong` para renombrar variables ligadas, y reconstrucción de los cuantificadores existenciales— es un patrón que se repetirá en otras partes del desarrollo donde se necesite compatibilidad entre diferentes versiones alpha-renombradas de una misma fórmula.

--
==========

-- SUB-ETAPA 2.1: EQUIVALENCIA Y REPRESENTABILIDAD DE BetaForm_sh

--
==========

```
-- 1. Función global para M0_term

def M0_term {s : Nat} (I B : Nat) : ArithTerm s :=

 ArithTerm.succ (ArithTerm.mul (ArithTerm.succ (natToTerm (s := s) I)) (natToTerm (s := s) B))


-- 2. Demostración global de clausura para M0_term

lemma isClosed_M0 {s : Nat} (I B : Nat) : isClosed (M0_term (s := s) I B) = true := by

 dsimp [M0_term, isClosed]

 rw [isClosed_natToTerm, isClosed_natToTerm]

 rfl


-- 3. Lema global de no-ocurrencia para M0_term

lemma NotOccur_M0 {s : Nat} (I B : Nat) (w : Nat) : NotOccur w (M0_term (s := s) I B) :=

 NotOccur_of_isClosed (M0_term I B) (isClosed_M0 I B) w


-- 4. Equivalencia alpha para BetaForm vs BetaForm_sh


lemma beta_sh_equivalence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

   (A B I : Nat) (R : ArithTerm s) (hR : isClosed R = true) :

   substForm 3 R (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm))) ∈ Δ ↔

   substForm 3 R (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) ∈ Δ := by


 have h_lhs : substForm 3 R (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm))) =

    exists_form 4 (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) (M0_term (s := s) I B)) R))
```

(exists_form 5 (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R (ArithTerm.succ (ArithTerm.var 5))))))) := by

dsimp [BetaForm, M0_term, exists_form, ArithForm.exists_form, and_form, lt_form, substForm, substTerm]

rw [substTerm_fresh (natToTerm A) 1 (natToTerm B) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 1),

substTerm_fresh (natToTerm A) 2 (natToTerm I) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2),

substTerm_fresh (natToTerm A) 3 R (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3),

substTerm_fresh (natToTerm B) 2 (natToTerm I) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 2),

substTerm_fresh (natToTerm B) 3 R (NotOccur_of_isClosed _ (isClosed_natToTerm B) 3),

substTerm_fresh (natToTerm I) 3 R (NotOccur_of_isClosed _ (isClosed_natToTerm I) 3)]

have h_rhs : substForm 3 R (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) =

exists_form 8 (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) (M0_term (s := s) I B)) R))

(exists_form 9 (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R (ArithTerm.succ (ArithTerm.var 9)))))) := by

dsimp [BetaForm_sh, M0_term, exists_form, ArithForm.exists_form, and_form, lt_form, substForm, substTerm]

rw [substTerm_fresh (natToTerm A) 1 (natToTerm B) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 1),

substTerm_fresh (natToTerm A) 2 (natToTerm I) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2),

substTerm_fresh (natToTerm A) 3 R (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3),

substTerm_fresh (natToTerm B) 2 (natToTerm I) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 2),

substTerm_fresh (natToTerm B) 3 R (NotOccur_of_isClosed _ (isClosed_natToTerm B) 3),

```
      substTerm_fresh (natToTerm I) 3 R (NotOccur_of_isClosed _ (isClosed_natToTerm
I) 3)]


  rw [h_lhs, h_rhs]


  let E5 := ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R (ArithTerm.succ
(ArithTerm.var 5)))

  have h_occ5 : NotOccurForm 9 E5 := by

    unfold NotOccurForm

    exact ⟨NotOccur_M0 I B 9, NotOccur.add _ _ (NotOccur_of_isClosed R hR 9)
(NotOccur.succ _ (NotOccur.var 5 (by decide)))⟩


  have h_eq59 := change_var_equivalence_strong Δ h_max h_con 5 9 E5 (by decide)
h_occ5 True.intro

  have h_subst59 : substForm 5 (ArithTerm.var 9) E5 = ArithForm.eq (M0_term (s := s)
I B) (ArithTerm.add R (ArithTerm.succ (ArithTerm.var 9))) := by

    dsimp [E5, substForm, substTerm]

    rw [substTerm_fresh (M0_term I B) 5 (ArithTerm.var 9) (NotOccur_M0 I B 5),

      substTerm_fresh R 5 (ArithTerm.var 9) (NotOccur_of_isClosed R hR 5)]

  rw [h_subst59] at h_eq59


  let lt5 := exists_form 5 E5

  let lt9 := exists_form 9 (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R
(ArithTerm.succ (ArithTerm.var 9))))


  have h_lt_eq : lt5 ∈ Δ ↔ lt9 ∈ Δ := h_eq59


  have h_nf_lt5 : NotFreeForm 4 lt5 := by

    dsimp [lt5, E5, exists_form, ArithForm.exists_form, NotFreeForm]

    exact ⟨NotOccur_M0 I B 4, NotOccur.add _ _ (NotOccur_of_isClosed R hR 4)
```

```
    (NotOccur.succ _ (NotOccur.var 5 (by decide)))⟩


  have h_nf_lt9 : NotFreeForm 8 lt9 := by
    dsimp [lt9, exists_form, ArithForm.exists_form, NotFreeForm]
    exact ⟨NotOccur_M0 I B 8, NotOccur.add _ _ (NotOccur_of_isClosed R hR 8)
      (NotOccur.succ _ (NotOccur.var 9 (by decide)))⟩


  constructor
  · intro h_in
    obtain ⟨q, hq⟩ := h_henkin 4 _ h_in
    change substForm 4 q.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) (M0_term (s := s) I B)) R)) lt5) ∈ Δ at hq
    have hq_simp : substForm 4 q.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) (M0_term (s := s) I B)) R)) lt5) =
              and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul q.val (M0_term (s := s) I B)) R)) lt5 := by
      unfold and_form
      rw [substForm_neg, substForm_impl, substForm_neg]
      rw [substForm_fresh_of_NotFreeForm lt5 4 q.val h_nf_lt5]
      rw [substForm_eq]
      dsimp [substTerm]
      rw [substTerm_fresh (natToTerm (s := s) A) 4 q.val (NotOccur_of_isClosed _ (isClosed_natToTerm A) 4),
        substTerm_fresh (M0_term (s := s) I B) 4 q.val (NotOccur_M0 I B 4),
        substTerm_fresh R 4 q.val (NotOccur_of_isClosed R hR 4)]
    rw [hq_simp] at hq
    have h_eq_q := and_elim_left_Δ h_max h_con _ _ hq
    have h_lt_q := and_elim_right_Δ h_max h_con _ _ hq
    have h_lt9_q := h_lt_eq.mp h_lt_q
```

```
  have h_and9 := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_eq_q) (SPA_Deduction_Internal.hyp Δ _ h_lt9_q)

  have h_and9_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and9

  have h_subst8 : substForm 8 q.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) (M0_term (s := s) I B)) R)) lt9) =

          and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul q.val (M0_term (s := s) I B)) R)) lt9 := by

    unfold and_form

    rw [substForm_neg, substForm_impl, substForm_neg]

    rw [substForm_fresh_of_NotFreeForm lt9 8 q.val h_nf_lt9]

    rw [substForm_eq]

    dsimp [substTerm]

    rw [substTerm_fresh (natToTerm (s := s) A) 8 q.val (NotOccur_of_isClosed _ (isClosed_natToTerm A) 8),

      substTerm_fresh (M0_term (s := s) I B) 8 q.val (NotOccur_M0 I B 8),

      substTerm_fresh R 8 q.val (NotOccur_of_isClosed R hR 8)]

  have h_subst8_in : substForm 8 q.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) (M0_term (s := s) I B)) R)) lt9) ∈ Δ := by

    rw [h_subst8]

    exact h_and9_in

  let φ8 := and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) (M0_term (s := s) I B)) R)) lt9

  have h_free : IsFreeFor 8 q.val φ8 := IsFreeFor_closed φ8 8 q

  have h_ex := exists_intro Δ 8 φ8 q.val h_free (SPA_Deduction_Internal.hyp Δ _ h_subst8_in)

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_ex


· intro h_in

  obtain ⟨q, hq⟩ := h_henkin 8 _ h_in
```

```
  change substForm 8 q.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) (M0_term (s := s) I B)) R)) lt9) ∈ Δ at hq

  have hq_simp : substForm 8 q.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) (M0_term (s := s) I B)) R)) lt9) =

        and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul q.val (M0_term (s := s) I B)) R)) lt9 := by

    unfold and_form

    rw [substForm_neg, substForm_impl, substForm_neg]

    rw [substForm_fresh_of_NotFreeForm lt9 8 q.val h_nf_lt9]

    rw [substForm_eq]

    dsimp [substTerm]

    rw [substTerm_fresh (natToTerm (s := s) A) 8 q.val (NotOccur_of_isClosed _ (isClosed_natToTerm A) 8),

      substTerm_fresh (M0_term (s := s) I B) 8 q.val (NotOccur_M0 I B 8),

      substTerm_fresh R 8 q.val (NotOccur_of_isClosed R hR 8)]

  rw [hq_simp] at hq

  have h_eq_q := and_elim_left_Δ h_max h_con _ _ hq

  have h_lt_q := and_elim_right_Δ h_max h_con _ _ hq

  have h_lt5_q := h_lt_eq.mpr h_lt_q

  have h_and5 := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_eq_q) (SPA_Deduction_Internal.hyp Δ _ h_lt5_q)

  have h_and5_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and5

  have h_subst4 : substForm 4 q.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) (M0_term (s := s) I B)) R)) lt5) =

        and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul q.val (M0_term (s := s) I B)) R)) lt5 := by

    unfold and_form

    rw [substForm_neg, substForm_impl, substForm_neg]

    rw [substForm_fresh_of_NotFreeForm lt5 4 q.val h_nf_lt5]

    rw [substForm_eq]
```

```
    dsimp [substTerm]

    rw [substTerm_fresh (natToTerm (s := s) A) 4 q.val (NotOccur_of_isClosed _ (isClosed_natToTerm A) 4),

      substTerm_fresh (M0_term (s := s) I B) 4 q.val (NotOccur_M0 I B 4),

      substTerm_fresh R 4 q.val (NotOccur_of_isClosed R hR 4)]

  have h_subst4_in : substForm 4 q.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) (M0_term (s := s) I B)) R)) lt5) ∈ Δ := by

    rw [h_subst4]

    exact h_and5_in

  let φ4 := and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 4) (M0_term (s := s) I B)) R)) lt5

  have h_free : IsFreeFor 4 q.val φ4 := IsFreeFor_closed φ4 4 q

  have h_ex := exists_intro Δ 4 φ4 q.val h_free (SPA_Deduction_Internal.hyp Δ _ h_subst4_in)

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_ex
```

Lema `beta_sh_representable`: Representabilidad de `BetaForm_sh`

Se demuestra que la versión alpha-renombrada `BetaForm_sh` hereda la propiedad de representabilidad de `BetaForm`. Este resultado es una consecuencia directa de dos ingredientes previos: el teorema `beta_representable` (que establece la representabilidad de `BetaForm`) y el lema `beta_sh_equivalence` (que establece la equivalencia lógica entre ambas versiones en Δ).

Formalmente, para cualesquiera $A, B, I, R \in \mathbb{N}$ con $R = \text{godelBeta}(A, B, I)$:

$$ \text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{I}, \overline{R}) \in \Delta $$

y para cualquier término cerrado R':

$$ \text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{I}, R') \in \Delta \implies R' = \overline{R} \in \Delta $$

Estructura de la demostración

La prueba es notablemente concisa gracias a la arquitectura modular:

1. Obtención del resultado original: Se invoca `beta_representable` con los mismos parámetros, que proporciona exactamente dos hechos:

- `h_orig.1`: `BetaForm` con los numerales de A, B, I, R pertenece a Δ.

- `h_orig.2`: La condición de funcionalidad para `BetaForm`.

2. Condición de gráfico para `BetaForm_sh`: Se aplica la dirección `mp` de `beta_sh_equivalence` al hecho `h_orig.1`. Dado que el cuarto argumento es $\overline{R}$ (que es un término cerrado, como requiere el lema de equivalencia), se obtiene directamente la pertenencia de `BetaForm_sh` a Δ.

3. Condición de funcionalidad para `BetaForm_sh`: Para un R' arbitrario tal que `BetaForm_sh` con R' pertenece a Δ, se aplica la dirección `mpr` de `beta_sh_equivalence` (con el término cerrado R') para obtener que `BetaForm` con

R' también pertenece a Δ. Entonces, la funcionalidad de `BetaForm` (`h_orig.2`) garantiza $R' = \overline{R} \in \Delta$.

Significado

`beta_sh_representable` ejemplifica el principio de que la representabilidad es invariante bajo alpha-conversión (renombramiento de variables ligadas). La demostración no requiere repetir el extenso argumento de `beta_representable`, sino que se apoya en la equivalencia lógica para transferir el resultado. Esta modularidad es esencial para la escalabilidad del desarrollo: una vez demostrada la representabilidad de una fórmula, cualquier versión alpha-equivalente es automáticamente representable mediante un lema de equivalencia adecuado.

La disponibilidad de `BetaForm_sh` (con variables ligadas 8 y 9) es crucial para su uso dentro de `NumForm`, donde las variables 4 y 5 (utilizadas por `BetaForm`) entrarían en conflicto con las variables de inducción y de valores intermedios. La técnica de renombrar variables ligadas para evitar capturas es estándar en lógica, y este lema formaliza que dicha técnica es correcta en el contexto de la representabilidad en Δ.

lemma beta_sh_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

(h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

(A B I R : Nat) (h_R : R = godelBeta A B I) :

(substForm 3 (natToTerm (s := s) R) (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) ∈ Δ) ∧

(∀ (R' : TermClosed s),

substForm 3 R'.val (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) ∈ Δ →

ArithForm.eq R'.val (natToTerm (s := s) R) ∈ Δ) := by

have h_orig := beta_representable Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ A B I R h_R

constructor

· exact (beta_sh_equivalence Δ h_max h_con h_henkin A B I (natToTerm R) (isClosed_natToTerm R)).mp h_orig.1

· intro R' h_in

exact h_orig.2 R' ((beta_sh_equivalence Δ h_max h_con h_henkin A B I R'.val R'.property).mpr h_in)

Sub-etapa 2.2: Equivalencia y representabilidad de `PairForm_sh`

Se establece la equivalencia lógica entre `PairForm` (que usa variable ligada v_3) y `PairForm_sh` (que usa variable ligada v_{10}) en cualquier teoría Δ maximal consistente. Esta equivalencia es análoga a la de `BetaForm_sh` y sigue el mismo patrón de alpha-conversión.

Lema `subst_PairForm_sh`: Simplificación de la sustitución en `PairForm_sh`

Es el análogo exacto de `subst_PairForm` para la versión alpha-renombrada. Establece que al instanciar las variables libres v_0, v_1, v_2 con numerales $\overline{a}, \overline{b}$ y un término P, la fórmula `PairForm_sh` se reduce a la misma disyunción que `PairForm`, con la única diferencia de que la variable testigo de `lt_form` es 10 en lugar de 3:

$$ \text{PairForm_sh}(\overline{a}, \overline{b}, P) = (\overline{a} < \overline{b} \text{ (testigo 10)} \land P = \overline{b}^2 + \overline{a}) \lor (\neg(\overline{a} < \overline{b}) \land P = \overline{a}^2 + \overline{a} + \overline{b}) $$

La demostración es idéntica a la de `subst_PairForm`, reemplazando la variable 3 por 10 y utilizando las mismas propiedades de frescura de los numerales.

Lema `pair_sh_equivalence`: Equivalencia entre `PairForm` y `PairForm_sh`

Establece que para parámetros fijos $a, b \in \mathbb{N}$ y un término cerrado P, las instancias de `PairForm` y `PairForm_sh` son equivalentes en Δ:

$$ \Delta \vdash \text{PairForm}(\overline{a}, \overline{b}, P) \iff \Delta \vdash \text{PairForm_sh}(\overline{a}, \overline{b}, P) $$

Estructura de la demostración

1. Simplificación de ambas fórmulas: Aplicando `subst_PairForm` y `subst_PairForm_sh`, ambas fórmulas se reducen a disyunciones con la misma estructura, difiriendo solo en el índice de la variable testigo de `lt_form` (3 versus 10):

$$ (\overline{a} <_3 \overline{b} \land P = \overline{b}^2 + \overline{a}) \lor (\neg(\overline{a} <_3 \overline{b}) \land P = \overline{a}^2 + \overline{a} + \overline{b}) $$

$$ (\overline{a} <_{10} \overline{b} \land P = \overline{b}^2 + \overline{a}) \lor (\neg(\overline{a} <_{10} \overline{b}) \land P = \overline{a}^2 + \overline{a} + \overline{b}) $$

2. Equivalencia de las fórmulas de orden: Se aplica `change_var_equivalence_strong` para establecer la equivalencia entre $\overline{a} <_3 \overline{b}$ y $\overline{a} <_{10} \overline{b}$ en Δ. La verificación de las hipótesis requiere:

- v_{10} no ocurre libre en la fórmula testigo (lo cual es cierto porque $\overline{a}$ y $\overline{b}$ son numerales cerrados, y v_3 está ligada).

- No hay conflictos de ligadura (la variable 3 no está ligada en un contexto superior, pues la fórmula de orden es atómica en este sentido).

3. Equivalencia de las negaciones: Usando maximalidad y consistencia, se demuestra que $\neg(\overline{a} <_3 \overline{b}) \in \Delta \iff \neg(\overline{a} <_{10} \overline{b}) \in \Delta$. La prueba utiliza el hecho de que si una de las fórmulas de orden no está en Δ, su negación sí lo está (por maximalidad), y la equivalencia del paso anterior fuerza que la otra fórmula de orden tampoco esté, por lo que su negación también está.

4. Equivalencia de las disyunciones: Se demuestran ambas direcciones mediante análisis de casos sobre cuál de las dos ramas de la disyunción pertenece a Δ:

- **Dirección $\Rightarrow$:** Si la primera rama ($\overline{a} <_3 \overline{b} \land \text{eq_left}$) está en Δ, por la equivalencia de orden se obtiene la primera rama con $<_{10}$. Si no, por maximalidad la negación de la primera rama está en Δ, y por la definición de $A \lor B$ como $\neg A \to B$, se deduce que la segunda rama está en Δ. Usando la equivalencia de las negaciones de orden, se transfiere a la segunda rama con $<_{10}$.

- **Dirección $\Leftarrow$:** Simétrica, usando las equivalencias en dirección contraria.

Significado

`pair_sh_equivalence` completa el conjunto de lemas de alpha-equivalencia necesarios para que `NumForm` pueda utilizar versiones renombradas de `BetaForm` y `PairForm` sin conflicto de variables. La técnica de demostración —reducción a formas explícitas, equivalencia de subfórmulas mediante `change_var_equivalence_strong`, y análisis de casos por maximalidad— es un patrón general aplicable a cualquier par de fórmulas que difieran solo en el nombre de sus variables ligadas.

La importancia de este resultado radica en que `PairForm_sh` aparece dentro de `IsSuccCode`, que a su vez es una subfórmula de `NumForm`. Sin el renombramiento de la variable ligada de 3 a 10, la variable 3 de `PairForm` entraría en conflicto con la variable 3 que `NumForm` utiliza para el argumento R en las llamadas a `BetaForm_sh`. Este tipo de gestión cuidadosa de las variables ligadas es característico de la aritmetización de la sintaxis, donde múltiples codificaciones y cuantificaciones anidadas requieren una disciplina rigurosa para evitar capturas accidentales.

```
--
================================================================

-- SUB-ETAPA 2.2: EQUIVALENCIA Y REPRESENTABILIDAD DE PairForm_sh

--
================================================================


/-- Lema de sustitución para PairForm_sh (análogo a subst_PairForm) -/

lemma subst_PairForm_sh {s : Nat} (a b : Nat) (P : ArithTerm s) :

  substForm 2 P (substForm 1 (@natToTerm s b) (substForm 0 (@natToTerm s a) (PairForm_sh (s := s)))) =

  or_form
```

(and_form (lt_form (@natToTerm s a) (@natToTerm s b) 10)

(ArithForm.eq P (ArithTerm.add (ArithTerm.mul (@natToTerm s b) (@natToTerm s b)) (@natToTerm s a))))

(and_form (ArithForm.neg (lt_form (@natToTerm s a) (@natToTerm s b) 10))

(ArithForm.eq P (ArithTerm.add (ArithTerm.add (ArithTerm.mul (@natToTerm s a) (@natToTerm s a)) (@natToTerm s a)) (@natToTerm s b)))) := by

unfold PairForm_sh

dsimp [or_form, and_form, lt_form, exists_form, ArithForm.exists_form, substForm, substTerm]

have h_a0 := substTerm_fresh (@natToTerm s a) 1 (@natToTerm s b) (NotOccur_of_isClosed (@natToTerm s a) (isClosed_natToTerm a) 1)

have h_a1 := substTerm_fresh (@natToTerm s a) 2 P (NotOccur_of_isClosed (@natToTerm s a) (isClosed_natToTerm a) 2)

have h_b0 := substTerm_fresh (@natToTerm s b) 2 P (NotOccur_of_isClosed (@natToTerm s b) (isClosed_natToTerm b) 2)

rw [h_a0, h_a1, h_b0]

/-- Equivalencia α para PairForm vs PairForm_sh -/

lemma pair_sh_equivalence {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (_h_henkin : HenkinSet Δ)

(a b : Nat) (P : ArithTerm s) (_hP : isClosed P = true) :

substForm 2 P (substForm 1 (@natToTerm s b) (substForm 0 (@natToTerm s a) PairForm)) ∈ Δ ↔

substForm 2 P (substForm 1 (@natToTerm s b) (substForm 0 (@natToTerm s a) PairForm_sh)) ∈ Δ := by

rw [subst_PairForm a b P, subst_PairForm_sh a b P]

-- Abreviaturas para las partes que NO cambian

```
 let eq_left := ArithForm.eq P (ArithTerm.add (ArithTerm.mul (@natToTerm s b) (@natToTerm s b)) (@natToTerm s a))

 let eq_right := ArithForm.eq P (ArithTerm.add (ArithTerm.add (ArithTerm.mul (@natToTerm s a) (@natToTerm s a)) (@natToTerm s a)) (@natToTerm s b))

 let lt3 := lt_form (@natToTerm s a) (@natToTerm s b) 3

 let lt10 := lt_form (@natToTerm s a) (@natToTerm s b) 10

 -- Equivalencia de lt_form ... 3 y lt_form ... 10 vía change_var_equivalence_strong

 have h_lt_eq : lt3 ∈ Δ ↔ lt10 ∈ Δ := by

  dsimp [lt3, lt10, lt_form, exists_form, ArithForm.exists_form]

  let ψ := ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var 3)))

  have h_occ : NotOccurForm 10 ψ := by

   dsimp [ψ, NotOccurForm]

   exact ⟨NotOccur_of_isClosed (@natToTerm s b) (isClosed_natToTerm b) 10,

      NotOccur.add _ _ (NotOccur_of_isClosed (@natToTerm s a) (isClosed_natToTerm a) 10)

       (NotOccur.succ _ (NotOccur.var 3 (by decide)))⟩

  have h_bound : NoBoundVar 3 ψ := True.intro

  have h_equiv := change_var_equivalence_strong Δ h_max h_con 3 10 ψ (by decide) h_occ h_bound

  have h_subst : substForm 3 (ArithTerm.var 10) ψ = ArithForm.eq (@natToTerm s b) (ArithTerm.add (@natToTerm s a) (ArithTerm.succ (ArithTerm.var 10))) := by

   dsimp [ψ, substForm, substTerm]

   rw [substTerm_fresh (@natToTerm s b) 3 (ArithTerm.var 10) (NotOccur_of_isClosed _ (isClosed_natToTerm b) 3),

     substTerm_fresh (@natToTerm s a) 3 (ArithTerm.var 10) (NotOccur_of_isClosed _ (isClosed_natToTerm a) 3)]

  rw [h_subst] at h_equiv

  exact h_equiv

 -- Equivalencia de las negaciones por maximalidad

 have h_neg_lt_eq : ArithForm.neg lt3 ∈ Δ ↔ ArithForm.neg lt10 ∈ Δ := by
```

```
  constructor
  · intro h_neg3
    by_contra h_not_neg10
    have h_lt10_in : lt10 ∈ Δ := by
      cases h_max lt10 with
      | inl h => exact h
      | inr h => exact absurd h h_not_neg10
    have h_lt3_in : lt3 ∈ Δ := h_lt_eq.mpr h_lt10_in
    exact h_con (SPA_Deduction_Internal.neg_elim Δ lt3
      (SPA_Deduction_Internal.hyp Δ _ h_lt3_in)
      (SPA_Deduction_Internal.hyp Δ _ h_neg3))
  · intro h_neg10
    by_contra h_not_neg3
    have h_lt3_in : lt3 ∈ Δ := by
      cases h_max lt3 with
      | inl h => exact h
      | inr h => exact absurd h h_not_neg3
    have h_lt10_in : lt10 ∈ Δ := h_lt_eq.mp h_lt3_in
    exact h_con (SPA_Deduction_Internal.neg_elim Δ lt10
      (SPA_Deduction_Internal.hyp Δ _ h_lt10_in)
      (SPA_Deduction_Internal.hyp Δ _ h_neg10))
-- Transferir la disyunción completa
constructor
· -- Dirección →
  intro h_or
  by_cases h_X3 : and_form lt3 eq_left ∈ Δ
  · -- Caso X3 ∈ Δ: transferir a X10
    have h_lt3_in := and_elim_left_Δ h_max h_con lt3 eq_left h_X3
```

```
    have h_eq_in := and_elim_right_Δ h_max h_con lt3 eq_left h_X3
    have h_lt10_in : lt10 ∈ Δ := h_lt_eq.mp h_lt3_in
    have h_X10 : and_form lt10 eq_left ∈ Δ :=
      maximal_consistent_closed_under_deduction Δ h_max h_con _
       (and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_lt10_in) (SPA_Deduction_Internal.hyp Δ _ h_eq_in))
    exact or_intro_left_Δ h_max h_con _ _ h_X10
  · -- Caso X3 ∉ Δ: entonces neg X3 ∈ Δ, y por or_form, Y3 ∈ Δ
    have h_neg_X3 : ArithForm.neg (and_form lt3 eq_left) ∈ Δ := by
      cases h_max (and_form lt3 eq_left) with
      | inl h => contradiction
      | inr h => exact h
    have h_Y3 : and_form (ArithForm.neg lt3) eq_right ∈ Δ :=
      maximal_consistent_imp_closed Δ h_max h_con _ _ h_neg_X3 h_or
    have h_neg_lt3_in := and_elim_left_Δ h_max h_con (ArithForm.neg lt3) eq_right h_Y3
    have h_eq_in := and_elim_right_Δ h_max h_con (ArithForm.neg lt3) eq_right h_Y3
    have h_neg_lt10_in : ArithForm.neg lt10 ∈ Δ := h_neg_lt_eq.mp h_neg_lt3_in
    have h_Y10 : and_form (ArithForm.neg lt10) eq_right ∈ Δ :=
      maximal_consistent_closed_under_deduction Δ h_max h_con _
       (and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_neg_lt10_in) (SPA_Deduction_Internal.hyp Δ _ h_eq_in))
    exact or_intro_right_Δ h_max h_con _ _ h_Y10
 · -- Dirección ←
  intro h_or
  by_cases h_X10 : and_form lt10 eq_left ∈ Δ
  · -- Caso X10 ∈ Δ: transferir a X3
    have h_lt10_in := and_elim_left_Δ h_max h_con lt10 eq_left h_X10
    have h_eq_in := and_elim_right_Δ h_max h_con lt10 eq_left h_X10
```

```
    have h_lt3_in : lt3 ∈ Δ := h_lt_eq.mpr h_lt10_in
    have h_X3 : and_form lt3 eq_left ∈ Δ :=
      maximal_consistent_closed_under_deduction Δ h_max h_con _
       (and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_lt3_in) (SPA_Deduction_Internal.hyp Δ _ h_eq_in))
    exact or_intro_left_Δ h_max h_con _ _ h_X3
  · -- Caso X10 ∉ Δ: entonces neg X10 ∈ Δ, y por or_form, Y10 ∈ Δ
    have h_neg_X10 : ArithForm.neg (and_form lt10 eq_left) ∈ Δ := by
      cases h_max (and_form lt10 eq_left) with
      | inl h => contradiction
      | inr h => exact h
    have h_Y10 : and_form (ArithForm.neg lt10) eq_right ∈ Δ :=
      maximal_consistent_imp_closed Δ h_max h_con _ _ h_neg_X10 h_or
    have h_neg_lt10_in := and_elim_left_Δ h_max h_con (ArithForm.neg lt10) eq_right h_Y10
    have h_eq_in := and_elim_right_Δ h_max h_con (ArithForm.neg lt10) eq_right h_Y10
    have h_neg_lt3_in : ArithForm.neg lt3 ∈ Δ := h_neg_lt_eq.mpr h_neg_lt10_in
    have h_Y3 : and_form (ArithForm.neg lt3) eq_right ∈ Δ :=
      maximal_consistent_closed_under_deduction Δ h_max h_con _
       (and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_neg_lt3_in) (SPA_Deduction_Internal.hyp Δ _ h_eq_in))
    exact or_intro_right_Δ h_max h_con _ _ h_Y3
```

Lemas `pair_sh_representable` e `IsSuccCode_representable`

Se demuestran dos resultados de representabilidad que completan la infraestructura necesaria para `NumForm`: la versión alpha-renombrada de `PairForm` es representable, y la fórmula `IsSuccCode` que expresa la relación entre códigos de términos sucesores también es representable.

Lema `pair_sh_representable`: Representabilidad de `PairForm_sh`

Establece que `PairForm_sh` hereda la representabilidad de `PairForm` mediante la equivalencia lógica demostrada en `pair_sh_equivalence`. Para cualesquiera $a, b \in \mathbb{N}$:

$$ \text{PairForm_sh}(\overline{a}, \overline{b}, \overline{\text{pair}(a, b)}) \in \Delta $$

y para cualquier término cerrado P:

$$ \text{PairForm_sh}(\overline{a}, \overline{b}, P) \in \Delta \implies P = \overline{\text{pair}(a, b)} \in \Delta $$

La demostración es directa: se invoca `pair_representable` para obtener el resultado original, y se aplica `pair_sh_equivalence` (en la dirección `mp` para el gráfico y `mpr` para la funcionalidad) para transferirlo a `PairForm_sh`. La estructura es idéntica a la de `beta_sh_representable`.

Lema `IsSuccCode_representable`: Representabilidad de la relación de códigos sucesores

Establece que la fórmula `IsSuccCode` representa correctamente la relación entre el código de un término v y el código de su sucesor $S(v)$. Recordemos que:

$$ \text{IsSuccCode}(v, v') := \exists p\ (\text{PairForm_sh}(\overline{4}, v, p) \land v' = S(p)) $$

Para cualesquiera $v, v' \in \mathbb{N}$ con $v' = \text{pair}(4, v) + 1$ (el código del sucesor según la codificación de Gödel):

1. **Gráfico:** $\Delta \vdash \text{IsSuccCode}(\overline{v}, \overline{v'})$

2. **Funcionalidad:** Si $\Delta \vdash \text{IsSuccCode}(\overline{v}, V')$, entonces $\Delta \vdash V' = \overline{v'}$

Estructura de la demostración

1. Construcción del testigo: Se define $p := \text{pair}(4, v)$. Por la representabilidad de `PairForm_sh` (`pair_sh_representable`), se tiene $\text{PairForm_sh}(\overline{4}, \overline{v}, \overline{p}) \in \Delta$.

2. Igualdad del sucesor: Dado que $v' = p + 1$, se tiene $\overline{v'} = S(\overline{p})$ (por definición de `natToTerm`). Esta igualdad es reflexiva en Δ.

3. Condición de gráfico: Se combinan la igualdad de `PairForm_sh` y la igualdad del sucesor mediante `and_intro`. Se define el cuerpo `body7` como la conjunción de ambas, y se verifica que al sustituir v_7 por $\overline{p}$ se obtiene exactamente la conjunción anterior. Se aplica `exists_intro` con testigo $\overline{p}$ para obtener `IsSuccCode` en Δ.

4. Verificación de frescura: Una parte considerable de la demostración se dedica a verificar que las variables 5, 6, 7 no ocurren libres en `PairForm_sh` (y en sus versiones con sustituciones parciales). Estas verificaciones, aunque tediosas, son necesarias para garantizar que las sustituciones y conmutaciones realizadas en la prueba son válidas. Los lemas `h_occ5_pf`, `h_occ6_pf`, `h_occ7_pf` establecen estas condiciones de no ocurrencia, y se propagan a través de las sustituciones mediante `NotOccurForm_subst_other`.

5. Condición de funcionalidad: Dado un término cerrado V' tal que $\text{IsSuccCode}(\overline{v}, V') \in \Delta$, por la propiedad de Henkin existe un testigo P para la variable v_7. Simplificando la sustitución (usando los lemas de frescura), se obtiene:

- $\text{PairForm_sh}(\overline{4}, \overline{v}, P) \in \Delta$

- $V' = S(P) \in \Delta$

Por la funcionalidad de `PairForm_sh`, $P = \overline{p} \in \Delta$. Por congruencia del sucesor y transitividad con la igualdad $\overline{v'} = S(\overline{p})$, se obtiene $V' = \overline{v'} \in \Delta$.

Significado

`IsSuccCode_representable` es el último eslabón en la cadena de representabilidad antes de abordar `NumForm`. Esta fórmula captura la operación sintáctica fundamental de construir el código del sucesor de un término a partir del código del término original. La codificación utiliza la función de emparejamiento con la etiqueta 4 (correspondiente al constructor `succ` en `serializeTerm`) y el esquema `encodeSeq` (que añade $+1$ para distinguir de la lista vacía).

La combinación de `beta_sh_representable` e `IsSuccCode_representable` proporciona todos los ingredientes necesarios para demostrar que `NumForm` es representable. La idea es que `NumForm` expresa internamente la construcción inductiva de los numerales:

$$ \text{NumForm}(w, z) \equiv \exists a, b\ \text{secuencia}(a, b, w, z) $$

donde $\text{secuencia}(a, b, w, z)$ verifica que la secuencia codificada por (a, b) comienza con el código de `zero`, termina con z en la posición w, y cada paso respeta la relación `IsSuccCode`. La representabilidad de cada componente (la función beta y la relación de códigos sucesores) allana el camino para la demostración final.

lemma pair_sh_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

(h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

(a b : Nat) :

(substForm 2 (@natToTerm s (pair a b)) (substForm 1 (@natToTerm s b) (substForm 0 (@natToTerm s a) PairForm_sh)) ∈ Δ) ∧

(∀ (P : TermClosed s),

substForm 2 P.val (substForm 1 (@natToTerm s b) (substForm 0 (@natToTerm s a) PairForm_sh)) ∈ Δ →

ArithForm.eq P.val (@natToTerm s (pair a b)) ∈ Δ) := by

have h_orig := pair_representable Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ a b

have h_eq_pair := pair_sh_equivalence Δ h_max h_con h_henkin a b (@natToTerm s (pair a b)) (isClosed_natToTerm (pair a b))

constructor

· exact h_eq_pair.mp h_orig.1

· intro P h_in

have h_eq_P := pair_sh_equivalence Δ h_max h_con h_henkin a b P.val P.property

exact h_orig.2 P (h_eq_P.mpr h_in)

lemma IsSuccCode_representable {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

(h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

(v v' : Nat) (h_v' : v' = pair 4 v + 1) :

(substForm 6 (natToTerm (s := s) v') (substForm 5 (natToTerm (s := s) v) IsSuccCode) ∈ Δ) ∧

(∀ (V' : TermClosed s),

substForm 6 V'.val (substForm 5 (natToTerm (s := s) v) IsSuccCode) ∈ Δ →

ArithForm.eq V'.val (natToTerm (s := s) v') ∈ Δ) := by

let p := pair 4 v

have h_pair_rep := pair_sh_representable Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ 4 v

have h_graph_pair : substForm 2 (natToTerm (s := s) p) (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)) ∈ Δ := h_pair_rep.1

have h_v'_term : natToTerm (s := s) v' = ArithTerm.succ (natToTerm (s := s) p) := by

dsimp [p]

rw [h_v']

```
  rfl


 have h_eq_succ : ArithForm.eq (natToTerm (s := s) v') (ArithTerm.succ (natToTerm (s := s) p)) ∈ Δ := by

  rw [← h_v'_term]

  exact eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) v')


 have h_and_ded := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ h_graph_pair) (SPA_Deduction_Internal.hyp Δ _ h_eq_succ)

 have h_and_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and_ded


 let body7 := and_form (substForm 2 (ArithTerm.var (s := s) 7) (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh))) (ArithForm.eq (natToTerm (s := s) v') (ArithTerm.succ (ArithTerm.var (s := s) 7)))


 -- Demostraciones explícitas de no-ocurrencia para PairForm_sh

 have h_occ7_pf : NotOccurForm 7 (PairForm_sh (s := s)) := by

  dsimp [PairForm_sh, or_form, and_form, lt_form, exists_form, ArithForm.exists_form, NotOccurForm]

  refine ⟨⟨by decide, ⟨NotOccur.var 1 (by decide), NotOccur.add _ _ (NotOccur.var 0 (by decide)) (NotOccur.succ _ (NotOccur.var 10 (by decide)))⟩⟩,

        ⟨NotOccur.var 2 (by decide), NotOccur.add _ _ (NotOccur.mul _ _ (NotOccur.var 1 (by decide)) (NotOccur.var 1 (by decide))) (NotOccur.var 0 (by decide))⟩⟩,

       ⟨⟨by decide, ⟨NotOccur.var 1 (by decide), NotOccur.add _ _ (NotOccur.var 0 (by decide)) (NotOccur.succ _ (NotOccur.var 10 (by decide)))⟩⟩,

        ⟨NotOccur.var 2 (by decide), NotOccur.add _ _ (NotOccur.add _ _ (NotOccur.mul _ _ (NotOccur.var 0 (by decide)) (NotOccur.var 0 (by decide))) (NotOccur.var 0 (by decide))) (NotOccur.var 1 (by decide))⟩⟩⟩


 have h_occ5_pf : NotOccurForm 5 (PairForm_sh (s := s)) := by

  dsimp [PairForm_sh, or_form, and_form, lt_form, exists_form, ArithForm.exists_form, NotOccurForm]
```

refine ⟨⟨by decide, ⟨NotOccur.var 1 (by decide), NotOccur.add _ _ (NotOccur.var 0 (by decide)) (NotOccur.succ _ (NotOccur.var 10 (by decide)))⟩⟩,

⟨NotOccur.var 2 (by decide), NotOccur.add _ _ (NotOccur.mul _ _ (NotOccur.var 1 (by decide)) (NotOccur.var 1 (by decide))) (NotOccur.var 0 (by decide))⟩⟩,

⟨⟨by decide, ⟨NotOccur.var 1 (by decide), NotOccur.add _ _ (NotOccur.var 0 (by decide)) (NotOccur.succ _ (NotOccur.var 10 (by decide)))⟩⟩,

⟨NotOccur.var 2 (by decide), NotOccur.add _ _ (NotOccur.add _ _ (NotOccur.mul _ _ (NotOccur.var 0 (by decide)) (NotOccur.var 0 (by decide))) (NotOccur.var 0 (by decide))) (NotOccur.var 1 (by decide))⟩⟩⟩

have h_occ6_pf : NotOccurForm 6 (PairForm_sh (s := s)) := by

dsimp [PairForm_sh, or_form, and_form, lt_form, exists_form, ArithForm.exists_form, NotOccurForm]

refine ⟨⟨by decide, ⟨NotOccur.var 1 (by decide), NotOccur.add _ _ (NotOccur.var 0 (by decide)) (NotOccur.succ _ (NotOccur.var 10 (by decide)))⟩⟩,

⟨NotOccur.var 2 (by decide), NotOccur.add _ _ (NotOccur.mul _ _ (NotOccur.var 1 (by decide)) (NotOccur.var 1 (by decide))) (NotOccur.var 0 (by decide))⟩⟩,

⟨⟨by decide, ⟨NotOccur.var 1 (by decide), NotOccur.add _ _ (NotOccur.var 0 (by decide)) (NotOccur.succ _ (NotOccur.var 10 (by decide)))⟩⟩,

⟨NotOccur.var 2 (by decide), NotOccur.add _ _ (NotOccur.add _ _ (NotOccur.mul _ _ (NotOccur.var 0 (by decide)) (NotOccur.var 0 (by decide))) (NotOccur.var 0 (by decide))) (NotOccur.var 1 (by decide))⟩⟩⟩

have h_occ7_sub0 : NotOccurForm 7 (substForm 0 (natToTerm (s := s) 4) PairForm_sh) :=

NotOccurForm_subst_other PairForm_sh 0 7 (natToTerm 4) h_occ7_pf (NotOccur_of_isClosed _ (isClosed_natToTerm 4) 7)

have h_occ7_sub1 : NotOccurForm 7 (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)) :=

NotOccurForm_subst_other _ 1 7 (natToTerm v) h_occ7_sub0 (NotOccur_of_isClosed _ (isClosed_natToTerm v) 7)

have h_occ5_sub0 : NotOccurForm 5 (substForm 0 (natToTerm (s := s) 4) PairForm_sh) :=

NotOccurForm_subst_other PairForm_sh 0 5 (natToTerm 4) h_occ5_pf (NotOccur_of_isClosed _ (isClosed_natToTerm 4) 5)

have h_occ6_sub0 : NotOccurForm 6 (substForm 0 (natToTerm (s := s) 4) PairForm_sh) :=

NotOccurForm_subst_other PairForm_sh 0 6 (natToTerm 4) h_occ6_pf (NotOccur_of_isClosed _ (isClosed_natToTerm 4) 6)

have h_occ6_sub1 : NotOccurForm 6 (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)) :=

NotOccurForm_subst_other _ 1 6 (natToTerm v) h_occ6_sub0 (NotOccur_of_isClosed _ (isClosed_natToTerm v) 6)

have h_subst7 : substForm 7 (natToTerm (s := s) p) body7 =

and_form (substForm 2 (natToTerm (s := s) p) (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)))

(ArithForm.eq (natToTerm (s := s) v') (ArithTerm.succ (natToTerm (s := s) p))) := by

dsimp [body7, and_form, substForm]

rw [substForm_substForm_term_comm (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)) 2 7 (natToTerm (s := s) p) h_occ7_sub1]

rw [substTerm_fresh (natToTerm (s := s) v') 7 (natToTerm (s := s) p) (NotOccur_of_isClosed _ (isClosed_natToTerm v') 7)]

dsimp [substTerm]

have h_and_in_rew : substForm 7 (natToTerm (s := s) p) body7 ∈ Δ := by

rwa [h_subst7]

have h_free7 : IsFreeFor 7 (natToTerm (s := s) p) body7 :=

```
  IsFreeFor_closed body7 7 ⟨natToTerm (s := s) p, isClosed_natToTerm p⟩

 have h_ex7 := exists_intro Δ 7 body7 (natToTerm (s := s) p) h_free7 (SPA_Deduction_Internal.hyp Δ _ h_and_in_rew)

 have h_ex7_in : exists_form 7 body7 ∈ Δ := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_ex7


 have h_goal1 : substForm 6 (natToTerm (s := s) v') (substForm 5 (natToTerm (s := s) v) IsSuccCode) = exists_form 7 body7 := by

  dsimp [IsSuccCode, body7, exists_form, ArithForm.exists_form, and_form, substForm]

  rw [substForm_comm (substForm 1 (ArithTerm.var (s := s) 5) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)) 5 2 (natToTerm (s := s) v) (ArithTerm.var (s := s) 7) (by decide)

    (NotOccur_of_isClosed _ (isClosed_natToTerm v) 2) (NotOccur.var 7 (by decide))]

  rw [substTerm_fresh (ArithTerm.var (s := s) 7) 5 (natToTerm (s := s) v) (NotOccur.var 7 (by decide))]

  rw [substForm_substForm_term_comm (substForm 0 (natToTerm (s := s) 4) PairForm_sh) 1 5 (natToTerm (s := s) v) h_occ5_sub0]

  rw [substForm_comm (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)) 6 2 (natToTerm (s := s) v') (ArithTerm.var (s := s) 7) (by decide)

    (NotOccur_of_isClosed _ (isClosed_natToTerm v') 2) (NotOccur.var 7 (by decide))]

  rw [substTerm_fresh (ArithTerm.var (s := s) 7) 6 (natToTerm (s := s) v') (NotOccur.var 7 (by decide))]

  rw [substForm_fresh_of_NotFreeForm _ 6 (natToTerm (s := s) v') (NotFreeForm_of_NotOccurForm 6 _ h_occ6_sub1)]

  dsimp [substTerm]


 constructor

 · -- ===== GRÁFICO =====

  rw [h_goal1]

  exact h_ex7_in
```

· -- ===== FUNCIONALIDAD =====

intro V' h_in

let body7_V' := and_form (substForm 2 (ArithTerm.var (s := s) 7) (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh))) (ArithForm.eq V'.val (ArithTerm.succ (ArithTerm.var (s := s) 7)))

have h_goal_V' : substForm 6 V'.val (substForm 5 (natToTerm (s := s) v) IsSuccCode) = exists_form 7 body7_V' := by

dsimp [IsSuccCode, body7_V', exists_form, ArithForm.exists_form, and_form, substForm]

rw [substForm_comm (substForm 1 (ArithTerm.var (s := s) 5) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)) 5 2 (natToTerm (s := s) v) (ArithTerm.var (s := s) 7) (by decide)

(NotOccur_of_isClosed _ (isClosed_natToTerm v) 2) (NotOccur.var 7 (by decide))]

rw [substTerm_fresh (ArithTerm.var (s := s) 7) 5 (natToTerm (s := s) v) (NotOccur.var 7 (by decide))]

rw [substForm_substForm_term_comm (substForm 0 (natToTerm (s := s) 4) PairForm_sh) 1 5 (natToTerm (s := s) v) h_occ5_sub0]

rw [substForm_comm (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)) 6 2 V'.val (ArithTerm.var (s := s) 7) (by decide)

(NotOccur_of_isClosed V'.val V'.property 2) (NotOccur.var 7 (by decide))]

rw [substTerm_fresh (ArithTerm.var (s := s) 7) 6 V'.val (NotOccur.var 7 (by decide))]

rw [substForm_fresh_of_NotFreeForm _ 6 V'.val (NotFreeForm_of_NotOccurForm 6 _ h_occ6_sub1)]

dsimp [substTerm]

rw [h_goal_V'] at h_in

obtain ⟨P, hP⟩ := h_henkin 7 body7_V' h_in

have hP_simp : substForm 7 P.val body7_V' =

```
     and_form (substForm 2 P.val (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)))

          (ArithForm.eq V'.val (ArithTerm.succ P.val)) := by

    dsimp [body7_V', and_form, substForm]

    rw [substForm_substForm_term_comm (substForm 1 (natToTerm (s := s) v) (substForm 0 (natToTerm (s := s) 4) PairForm_sh)) 2 7 P.val h_occ7_sub1]

    rw [substTerm_fresh V'.val 7 P.val (NotOccur_of_isClosed V'.val V'.property 7)]

    dsimp [substTerm]


  rw [hP_simp] at hP


  have h_pair_P := and_elim_left_Δ h_max h_con _ _ hP

  have h_eq_V' := and_elim_right_Δ h_max h_con _ _ hP


  have h_P_eq := h_pair_rep.2 P h_pair_P

  have h_succ_P := eq_succ_Δ h_max h_con h_P_eq


  have h_refl_v' : ArithForm.eq (ArithTerm.succ (natToTerm (s := s) p)) (natToTerm (s := s) v') ∈ Δ := by

    rw [← h_v'_term]

    exact eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) v')


  have h_trans1 := eq_trans_Δ h_max h_con h_eq_V' h_succ_P

  exact eq_trans_Δ h_max h_con h_trans1 h_refl_v'
```

Lemas de frescura para `C1`: Variables 0 y 1 no ocurren libres

Se demuestran dos lemas que establecen que las variables v_0 y v_1 no ocurren libres en la subfórmula `C1` de `NumForm`, donde `C1` es la instancia de `BetaForm_sh` que expresa el caso base de la secuencia:

$$ \text{C1} := \text{BetaForm_sh}(\overline{a}, \overline{b}, \overline{0}, \overline{\text{zeroCode}}) $$

pero con las variables de parámetro a y b representadas como v_2 y v_3 en lugar de v_{12} y v_{13} (este cambio de numeración es interno a la implementación).

Lema `NotFreeForm_C1_0`: Variable 0 no es libre en `C1`

Establece que v_0 no ocurre libre en la subfórmula que expresa $\beta(a, b, 0) = \overline{\text{zeroCode}}$ dentro de `NumForm`.

Lema `NotFreeForm_C1_1`: Variable 1 no es libre en `C1`

Análogo para v_1.

Estructura de las demostraciones

Ambas demostraciones siguen el mismo patrón, expandiendo completamente las definiciones de `BetaForm_sh`, los cuantificadores existenciales, las conjunciones, y las sustituciones. Para cada variable (0 o 1), se verifica que:

1. **Variables ligadas internas:** Las variables 8 y 9 (ligadas por los cuantificadores existenciales de `BetaForm_sh`) no son 0 ni 1, por lo que los condicionales `if v = 8` y `if v = 9` en la definición de `NotFreeForm` toman la rama negativa, requiriendo la verificación explícita de no ocurrencia.

2. **Numerales cerrados:** Los numerales $\overline{0}$ y $\overline{\text{zeroCode}}$ son términos cerrados, por lo que ninguna variable (en particular 0 o 1) ocurre en ellos. Esto se justifica mediante `NotOccur_of_isClosed` con `isClosed_natToTerm`.

3. **Variables v_2 y v_3 (parámetros a y b):** Estas variables aparecen en las posiciones donde `BetaForm_sh` espera los parámetros de la función beta. Son distintas de 0 y 1, por lo que no generan ocurrencias libres de 0 o 1.

4. **Componentes de la fórmula:** Cada subcomponente (los términos de la igualdad, la multiplicación, el sucesor, la suma) se analiza verificando que ni 0 ni 1 aparecen en ellos, utilizando los constructores de `NotOccur` (`zero`, `succ`, `mul`, `add`, `var` con desigualdad).

Significado

Estos lemas, junto con `NotFreeForm_C2` (que trata las variables 0 y 1 en la subfórmula del elemento final), son necesarios para establecer que al sustituir las

variables libres v_0 y v_1 de `NumForm` por términos concretos, no se producen capturas indeseadas en las subfórmulas `C1` y `C2`.

Específicamente, cuando se quiera demostrar la condición de gráfico para `NumForm` —es decir, que $\text{NumForm}(\overline{w}, \overline{z})$ pertenece a Δ para los valores correctos—, será necesario instanciar v_0 con $\overline{w}$ y v_1 con $\overline{z}$. Los lemas de frescura garantizan que estas sustituciones no alteran las subfórmulas `C1` y `C2` de manera imprevista, ya que 0 y 1 no ocurren libres en ellas. Esto simplifica considerablemente las demostraciones de representabilidad, pues permite tratar cada subfórmula de `NumForm` de manera independiente.

```
--
============================================================

-- LEMAS DE FRESCURA PARA C1

--
============================================================


lemma NotFreeForm_C1_0 {s : Nat} :

   NotFreeForm 0 (substForm 3 (natToTerm (s := s) zeroCode)

     (substForm 2 (natToTerm (s := s) 0)

       (substForm 1 (ArithTerm.var 3)

         (substForm 0 (ArithTerm.var 2) BetaForm_sh)))) := by

  dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, NotFreeForm, substForm, substTerm, natToTerm]

  have h0zc : NotOccur 0 (natToTerm (s := s) zeroCode) := NotOccur_of_isClosed (natToTerm zeroCode) (isClosed_natToTerm zeroCode) 0

  have h08 : NotOccur (s := s) 0 (ArithTerm.var 8) := NotOccur.var 8 (by decide : 0 ≠ 8)

  have h09 : NotOccur (s := s) 0 (ArithTerm.var 9) := NotOccur.var 9 (by decide : 0 ≠ 9)

  refine ⟨⟨NotOccur.zero, ?_⟩, ?_⟩
```

```
  · apply NotOccur.add
    · apply NotOccur.mul
      · exact h08
      · apply NotOccur.succ
        apply NotOccur.mul
        · apply NotOccur.succ
          exact NotOccur.zero
        · exact h0zc
    · exact h0zc
  · refine ⟨?_, ?_⟩
    · apply NotOccur.succ
      apply NotOccur.mul
      · apply NotOccur.succ
        exact NotOccur.zero
      · exact h0zc
    · apply NotOccur.add
      · exact h0zc
      · apply NotOccur.succ
        exact h09


lemma NotFreeForm_C1_1 {s : Nat} :
    NotFreeForm 1 (substForm 3 (natToTerm (s := s) zeroCode)
      (substForm 2 (natToTerm (s := s) 0)
        (substForm 1 (ArithTerm.var 3)
          (substForm 0 (ArithTerm.var 2) BetaForm_sh)))) := by
  dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form,
NotFreeForm, substForm, substTerm, natToTerm]
  have h1zc : NotOccur 1 (natToTerm (s := s) zeroCode) := NotOccur_of_isClosed
(natToTerm zeroCode) (isClosed_natToTerm zeroCode) 1
```

```
have h18 : NotOccur (s := s) 1 (ArithTerm.var 8) := NotOccur.var 8 (by decide : 1 ≠ 8)
have h19 : NotOccur (s := s) 1 (ArithTerm.var 9) := NotOccur.var 9 (by decide : 1 ≠ 9)
refine ⟨⟨NotOccur.zero, ?_⟩, ?_⟩
· apply NotOccur.add
  · apply NotOccur.mul
    · exact h18
    · apply NotOccur.succ
      apply NotOccur.mul
      · apply NotOccur.succ
        exact NotOccur.zero
      · exact h1zc
  · exact h1zc
· refine ⟨?_, ?_⟩
  · apply NotOccur.succ
    apply NotOccur.mul
    · apply NotOccur.succ
      exact NotOccur.zero
    · exact h1zc
  · apply NotOccur.add
    · exact h1zc
    · apply NotOccur.succ
      exact h19
```

Lema `C2_subst_eq` y teorema `NumForm_subst_closed`

Se demuestran dos resultados técnicos que simplifican la aplicación de sustituciones a `NumForm`. El primero es un lema auxiliar de conmutación para la subfórmula `C2`; el segundo es el teorema principal que establece la forma explícita que adopta `NumForm` cuando sus variables libres v_0 y v_1 se instancian con términos cerrados.

Lema `C2_subst_eq`: Conmutación de sustituciones en la subfórmula del elemento final

Establece que al sustituir v_0 por un término cerrado w y v_1 por un término cerrado z en la subfórmula `C2` (que expresa $\beta(a, b, v_0) = v_1$ dentro de `NumForm`), el resultado es equivalente a sustituir directamente los argumentos de `BetaForm_sh`:

$$ \text{C2}[v_0 := w,\ v_1 := z] = \text{BetaForm_sh}(\overline{a}, \overline{b}, w, z) $$

donde $\overline{a}$ y $\overline{b}$ están representados por las variables v_{12} y v_{13} (parámetros de la secuencia). La demostración es directa: dado que w es cerrado, las sustituciones sobre v_0 y v_1 en los argumentos de `BetaForm_sh` se simplifican usando `substTerm_fresh`, ya que v_1 no ocurre en w y v_3 no ocurre en w.

Teorema `NumForm_subst_closed`: Simplificación de `NumForm` instanciada

Establece la forma canónica que adopta `NumForm` cuando sus variables libres v_0 y v_1 se sustituyen por términos cerrados w y z:

$$ \text{NumForm}[v_0 := w,\ v_1 := z] = \exists a, b\ ($$

$$ \beta(a, b, 0) = \overline{\text{zeroCode}} \ \land $$

$$ \beta(a, b, w) = z \ \land $$

$$ \forall k\ (k < w \to \exists v, v'\ (\beta(a, b, k) = v \land \beta(a, b, k+1) = v' \land \text{IsSuccCode}(v, v')))) $$

donde las variables de parámetro a y b son v_{12} y v_{13}, y las llamadas a `BetaForm_sh` están completamente instanciadas.

Estructura de la demostración

La prueba es una verificación sistemática de que las sustituciones de v_0 y v_1 sobre `NumForm` no afectan a las subfórmulas donde estas variables no aparecen libres, y actúan de manera precisa sobre las subfórmulas donde sí aparecen. Se demuestran ocho igualdades auxiliares:

1. `hC1_0` y `hC1_1`: Las sustituciones de v_0 y v_1 no afectan a `C1` (el caso base) porque ni v_0 ni v_1 ocurren libres en `C1` (lemas `NotFreeForm_C1_0` y `NotFreeForm_C1_1`). Se aplica `substForm_fresh_of_NotFreeForm`.

2. `hC2`: La sustitución sobre `C2` (el elemento final) se resuelve mediante el lema `C2_subst_eq`, que conmuta las sustituciones externas con las internas de `BetaForm_sh`.

3. `hC4_0`, `hC4_1`, `hC5_0`, `hC5_1`: Las sustituciones de v_0 y v_1 no afectan a las subfórmulas que expresan $\beta(a, b, k) = v$ y $\beta(a, b, k+1) = v'$, porque v_0 y v_1 no ocurren libres en ellas (lemas `NotFreeForm_C4` y `NotFreeForm_C5`).

4. `hC6_0` y `hC6_1`: Las sustituciones de v_0 y v_1 no afectan a `IsSuccCode`, porque ninguna de estas variables aparece en dicha fórmula.

5. Simplificación final: Tras aplicar todas las igualdades, se reescribe la sustitución de v_1 por z en la variable v_1 dentro de la condición de orden $k < w$, lo cual produce $k < w$ (sin cambios porque v_1 no aparece en w).

Significado

`NumForm_subst_closed` es el resultado técnico que permite manipular `NumForm` como una fórmula representadora estándar. Al instanciar las variables libres con términos cerrados (típicamente numerales), la fórmula adopta una forma donde todas las referencias a v_0 y v_1 han sido reemplazadas por los valores concretos, mientras que las variables ligadas y las subfórmulas no afectadas permanecen intactas.

Este teorema es indispensable para la demostración de que `NumForm` satisface las condiciones de `IsRepresentable`. En particular:

- Para la **condición de gráfico**, se instanciará $v_0 := \overline{n}$ y $v_1 := \overline{\text{código_de_}\overline{n}}$, y se deberá demostrar que la fórmula resultante pertenece a Δ. `NumForm_subst_closed` proporciona la forma explícita de esta instancia, reduciendo el problema a verificar las tres condiciones que definen la construcción inductiva de numerales.

- Para la **condición de funcionalidad**, se instanciará $v_0 := \overline{n}$ y v_1 con un término cerrado arbitrario Z, y se deberá demostrar $Z = \overline{\text{código_de_}\overline{n}}$. Nuevamente, la forma explícita proporcionada por el teorema permite descomponer la hipótesis en las tres condiciones, y utilizar la representabilidad de `BetaForm_sh` e `IsSuccCode` para forzar la unicidad.

La demostración, aunque extensa en la verificación de cada igualdad, sigue un patrón mecánico: para cada subfórmula, se invoca el lema de frescura correspondiente (demostrado previamente) que garantiza que la variable no aparece libre, y se aplica `substForm_fresh_of_NotFreeForm`. Esta arquitectura modular, donde las condiciones de frescura se demuestran una vez y se reutilizan, es la que hace manejable la complejidad de `NumForm`.

```
--
================================================================

-- LEMA AUXILIAR DE SUSTITUCIÓN PARA C2

--
================================================================


lemma C2_subst_eq {s : Nat} (w z : ArithTerm s) (hw : isClosed w = true) :

   substForm 1 z (substForm 0 w (substForm 3 (ArithTerm.var (s := s) 1)

     (substForm 2 (ArithTerm.var (s := s) 0)

       (substForm 1 (ArithTerm.var (s := s) 13)

         (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))) =

   substForm 3 z (substForm 2 w (substForm 1 (ArithTerm.var (s := s) 13) (substForm
0 (ArithTerm.var (s := s) 12) BetaForm_sh))) := by

  dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form,
substForm, substTerm]

  rw [substTerm_fresh w 1 z (NotOccur_of_isClosed w hw 1)]

  rw [substTerm_fresh w 3 z (NotOccur_of_isClosed w hw 3)]


--
================================================================

-- TEOREMA PRINCIPAL: NumForm_subst_closed
```

```
--
================================================================


lemma NumForm_subst_closed {s : Nat} (w z : ArithTerm s) (hw : isClosed w = true)
(_hz : isClosed z = true) :
  substForm 1 z (substForm 0 w (NumForm (s := s))) =
  exists_form 12 (exists_form 13 (
   and_form
    (substForm 3 (natToTerm (s := s) zeroCode)
     (substForm 2 (natToTerm (s := s) 0)
      (substForm 1 (ArithTerm.var (s := s) 13)
       (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))
    (and_form
     (substForm 3 z
      (substForm 2 w
       (substForm 1 (ArithTerm.var (s := s) 13)
        (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))
     (ArithForm.forall_form 4 (ArithForm.impl
      (lt_form (ArithTerm.var (s := s) 4) w 11)
      (exists_form 5 (exists_form 6 (
       and_form
        (substForm 3 (ArithTerm.var (s := s) 5)
         (substForm 2 (ArithTerm.var (s := s) 4)
          (substForm 1 (ArithTerm.var (s := s) 13)
           (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))
        (and_form
         (substForm 3 (ArithTerm.var (s := s) 6)
          (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))
           (substForm 1 (ArithTerm.var (s := s) 13)
```

```
                (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))
            IsSuccCode
          )
        )))
       ))
      )
  )) := by
 have hC1_0 : substForm 0 w
    (substForm 3 (natToTerm (s := s) zeroCode)
     (substForm 2 (natToTerm (s := s) 0)
      (substForm 1 (ArithTerm.var (s := s) 13)
       (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh)))) =
  substForm 3 (natToTerm (s := s) zeroCode)
   (substForm 2 (natToTerm (s := s) 0)
    (substForm 1 (ArithTerm.var (s := s) 13)
     (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))) :=
  substForm_fresh_of_NotFreeForm _ 0 w (NotFreeForm_C1 0 (by decide) (by
decide))


 have hC1_1 : substForm 1 z
    (substForm 3 (natToTerm (s := s) zeroCode)
     (substForm 2 (natToTerm (s := s) 0)
      (substForm 1 (ArithTerm.var (s := s) 13)
       (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh)))) =
  substForm 3 (natToTerm (s := s) zeroCode)
   (substForm 2 (natToTerm (s := s) 0)
    (substForm 1 (ArithTerm.var (s := s) 13)
     (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))) :=
```

```
  substForm_fresh_of_NotFreeForm _ 1 z (NotFreeForm_C1 1 (by decide) (by decide))


 have hC4_0 : substForm 0 w
    (substForm 3 (ArithTerm.var (s := s) 5)
     (substForm 2 (ArithTerm.var (s := s) 4)
      (substForm 1 (ArithTerm.var (s := s) 13)
       (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh)))) =
   substForm 3 (ArithTerm.var (s := s) 5)
    (substForm 2 (ArithTerm.var (s := s) 4)
     (substForm 1 (ArithTerm.var (s := s) 13)
      (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))) :=
   substForm_fresh_of_NotFreeForm _ 0 w (NotFreeForm_C4 0 (by decide) (by decide)
(by decide) (by decide) (by decide) (by decide))


 have hC4_1 : substForm 1 z
    (substForm 3 (ArithTerm.var (s := s) 5)
     (substForm 2 (ArithTerm.var (s := s) 4)
      (substForm 1 (ArithTerm.var (s := s) 13)
       (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh)))) =
   substForm 3 (ArithTerm.var (s := s) 5)
    (substForm 2 (ArithTerm.var (s := s) 4)
     (substForm 1 (ArithTerm.var (s := s) 13)
      (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))) :=
   substForm_fresh_of_NotFreeForm _ 1 z (NotFreeForm_C4 1 (by decide) (by decide)
(by decide) (by decide) (by decide) (by decide))


 have hC5_0 : substForm 0 w
    (substForm 3 (ArithTerm.var (s := s) 6)
     (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))
```

```
      (substForm 1 (ArithTerm.var (s := s) 13)
       (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh)))) =
  substForm 3 (ArithTerm.var (s := s) 6)
   (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))
    (substForm 1 (ArithTerm.var (s := s) 13)
     (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))) :=
  substForm_fresh_of_NotFreeForm _ 0 w (NotFreeForm_C5 0 (by decide) (by decide)
(by decide) (by decide) (by decide) (by decide))

 have hC5_1 : substForm 1 z
   (substForm 3 (ArithTerm.var (s := s) 6)
    (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))
     (substForm 1 (ArithTerm.var (s := s) 13)
      (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh)))) =
  substForm 3 (ArithTerm.var (s := s) 6)
   (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))
    (substForm 1 (ArithTerm.var (s := s) 13)
     (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))) :=
  substForm_fresh_of_NotFreeForm _ 1 z (NotFreeForm_C5 1 (by decide) (by decide)
(by decide) (by decide) (by decide) (by decide))

 have hC6_0 : substForm 0 w IsSuccCode = IsSuccCode :=
  substForm_fresh_of_NotFreeForm _ 0 w (NotFreeForm_IsSuccCode 0 (by decide)
(by decide))

 have hC6_1 : substForm 1 z IsSuccCode = IsSuccCode :=
  substForm_fresh_of_NotFreeForm _ 1 z (NotFreeForm_IsSuccCode 1 (by decide) (by
decide))

 have hC2 : substForm 1 z
```

```
    (substForm 0 w
     (substForm 3 (ArithTerm.var (s := s) 1)
      (substForm 2 (ArithTerm.var (s := s) 0)
       (substForm 1 (ArithTerm.var (s := s) 13)
        (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))) =
  substForm 3 z (substForm 2 w (substForm 1 (ArithTerm.var (s := s) 13) (substForm
0 (ArithTerm.var (s := s) 12) BetaForm_sh))) :=
  C2_subst_eq w z hw


 dsimp [NumForm, exists_form, ArithForm.exists_form, and_form, lt_form, substForm,
substTerm]
 rw [hC1_0, hC1_1, hC2, hC4_0, hC4_1, hC5_0, hC5_1, hC6_0, hC6_1]
 rw [substTerm_fresh w 1 z (NotOccur_of_isClosed w hw 1)]
```

Sucesión de códigos de numerales y lemas de frescura adicionales

Se definen la función `seq_code` que calcula externamente la sucesión de códigos de Gödel para los numerales de Peano, y se demuestran lemas de frescura para las versiones completamente instanciadas de `BetaForm_sh` que aparecerán en la demostración de representabilidad de `NumForm`.

Función `seq_code`: Códigos de los numerales

Define recursivamente la secuencia de códigos correspondientes a los numerales $\overline{0}, \overline{1}, \overline{2}, \ldots$:

$$ \text{seq_code}(0) := \text{zeroCode} = \text{encode}(\text{serializeTerm}(\text{zero})) $$

$$ \text{seq_code}(k+1) := \text{pair}(4, \text{seq_code}(k)) + 1 $$

Lema `seq_code_eq_cn`: Equivalencia con la codificación interna

Establece que `seq_code n` coincide exactamente con la codificación de Gödel del numeral $\overline{n}$:

$$ \forall n \in \mathbb{N},\ \text{seq_code}(n) = \text{encode}(\text{serializeTerm}(\text{natToTerm}_{(s:=s+1)}\ n)) $$

La demostración es por inducción sobre n: el caso base es por definición, y el paso inductivo utiliza la definición recursiva de `seq_code` junto con la hipótesis de inducción.

Intuición

Esta función proporciona el valor semántico que `NumForm` debe capturar internamente: el código de Gödel del numeral $\overline{n}$. La representabilidad de `NumForm` afirmará precisamente que $\text{NumForm}(\overline{n}, \overline{\text{seq_code}(n)})$ es demostrable en Δ, y que cualquier Z que satisfaga $\text{NumForm}(\overline{n}, Z)$ es demostrablemente igual a $\overline{\text{seq_code}(n)}$.

Lemas de frescura para `BetaForm_sh` instanciada

Se demuestran tres lemas que establecen condiciones de no ocurrencia para versiones de `BetaForm_sh` donde los argumentos A y B (y posiblemente I y R) han sido instanciados con numerales concretos.

Lema `NotOccurForm_Beta_subst_AB`: Frescura tras instanciar A y B

Establece que si w es una variable distinta de las variables libres de `BetaForm_sh` ($0,1,2,3$) y de sus variables ligadas ($8,9$), entonces w no ocurre libre en `BetaForm_sh` con los argumentos A y B instanciados por numerales:

$$ \forall A, B \in \mathbb{N},\ \forall w \notin \{0,1,2,3,8,9\},\ \text{NotFreeForm } w\ (\text{BetaForm_sh}(\overline{A}, \overline{B}, \cdot, \cdot)) $$

La demostración propaga la condición de frescura a través de las sustituciones usando `NotOccurForm_subst_other`, verificando que los numerales $\overline{A}$ y $\overline{B}$ son términos cerrados y por tanto no contienen a w.

Lema `NotFreeForm_C4_closed`: Frescura de `BetaForm_sh` completamente instanciada

Generaliza el lema anterior al caso en que los cuatro argumentos de `BetaForm_sh` (A, B, I, R) están instanciados con numerales concretos. Establece que cualquier variable v distinta de las variables libres y ligadas de `BetaForm_sh` no es libre en la fórmula resultante:

$$ \forall A, B, k, v_k \in \mathbb{N},\ \forall v \notin \{0,1,2,3,8,9\},\ \text{NotFreeForm } v\ (\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k}, \overline{v_k})) $$

Lema `NotFreeForm_C5_closed`: Frescura para el paso inductivo (versión numeral)

Similar a `NotFreeForm_C4_closed`, pero para la fórmula que aparece en el paso inductivo con índice $k+1$:

$$ \forall A, B, k, v_{k+1} \in \mathbb{N},\ \forall v \notin \{0,1,2,3,8,9\},\ \text{NotFreeForm } } v\ (\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k+1}, \overline{v_{k+1}})) $$

Lema `NotFreeForm_C5_var6`: Frescura con variable ligada en la posición R

Versión del lema anterior donde el cuarto argumento no es un numeral sino la variable ligada v_6 (que representa v' en `NumForm`). Establece que cualquier variable v distinta de $\{0,1,2,3,6,8,9\}$ no es libre en:

$$ \text{BetaForm_sh}(\overline{A}, \overline{B}, S(\overline{k}), v_6) $$

Significado

Estos lemas de frescura son los últimos ingredientes técnicos necesarios para la demostración de `NumForm_representable`. Su propósito es garantizar que cuando se realicen sustituciones y se apliquen los lemas de representabilidad de `BetaForm_sh` e `IsSuccCode`, las condiciones de no ocurrencia requeridas por dichos lemas se satisfacen automáticamente.

En particular, durante la demostración de la funcionalidad de `NumForm`, será necesario aplicar `beta_sh_representable` a versiones instanciadas de `BetaForm_sh` con diferentes combinaciones de argumentos (A, B fijos, I variando entre k y $k+1$, R variando entre v y v'). Los lemas de frescura aseguran que las variables ligadas de `NumForm` ($v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}$) no aparecen libres en estas instancias de `BetaForm_sh`, evitando capturas y permitiendo la aplicación segura de las reglas de sustitución y los lemas de representabilidad.

```
--
================================================================

-- PASO 1: SUCESIÓN DE CÓDIGOS DE NUMERALES Y SU CORRECCIÓN

--
================================================================


/-- Sucesión en Nat de los códigos de los numerales de Peano -/

def seq_code : Nat → Nat

 | 0 => zeroCode

 | k + 1 => pair 4 (seq_code k) + 1


/-- Teorema de equivalencia entre seq_code n y el código del numeral n -/

lemma seq_code_eq_cn {s : Nat} (n : Nat) :

   seq_code n = Encodable.encode (serializeTerm (natToTerm (s := s+1) n)) := by

 induction n with

 | zero => rfl

 | succ k ih =>

   dsimp [seq_code]
```

```
  rw [ih]
  rfl
```

```
--
================================================================
-- LEMAS AUXILIARES DE FRESCURA EN BetaForm_sh Y SUBFÓRMULAS
--
================================================================

lemma NotOccurForm_Beta_subst_AB {s : Nat} (A B : Nat) (w : Nat)
  (hw0 : w ≠ 0) (hw1 : w ≠ 1) (hw2 : w ≠ 2) (hw3 : w ≠ 3) (hw8 : w ≠ 8) (hw9 : w ≠ 9) :
  NotOccurForm w (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh)) := by
 apply NotOccurForm_subst_other
 · apply NotOccurForm_subst_other
  · dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, NotOccurForm]
    refine ⟨hw8, ⟨⟨NotOccur.var 0 hw0, ?_⟩, ⟨hw9, ⟨?_, ?_⟩⟩⟩⟩
    · apply NotOccur.add
      · apply NotOccur.mul
        · exact NotOccur.var 8 hw8
```

```
        · apply NotOccur.succ
          apply NotOccur.mul
          · apply NotOccur.succ
            exact NotOccur.var 2 hw2
          · exact NotOccur.var 1 hw1
        · exact NotOccur.var 3 hw3
      · apply NotOccur.succ
        apply NotOccur.mul
        · apply NotOccur.succ
          exact NotOccur.var 2 hw2
        · exact NotOccur.var 1 hw1
      · apply NotOccur.add
        · exact NotOccur.var 3 hw3
        · apply NotOccur.succ
          exact NotOccur.var 9 hw9
    · exact NotOccur_of_isClosed _ (isClosed_natToTerm A) w
  · exact NotOccur_of_isClosed _ (isClosed_natToTerm B) w

lemma NotFreeForm_C4_closed {s : Nat} (A B k vk : Nat) (v : Nat)
    (hv0 : v ≠ 0) (hv1 : v ≠ 1) (hv2 : v ≠ 2) (hv3 : v ≠ 3) (hv8 : v ≠ 8) (hv9 : v ≠ 9) :
    NotFreeForm v (substForm 3 (natToTerm (s := s) vk)
      (substForm 2 (natToTerm (s := s) k)
        (substForm 1 (natToTerm (s := s) B)
          (substForm 0 (natToTerm (s := s) A) BetaForm_sh)))) := by
  apply NotFreeForm_of_NotOccurForm
  apply NotOccurForm_subst_other
  · apply NotOccurForm_subst_other
    · apply NotOccurForm_subst_other
```

```
    · apply NotOccurForm_subst_other
     · dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form,
NotOccurForm]
       refine ⟨hv8, ⟨⟨NotOccur.var 0 hv0, ?_⟩, ⟨hv9, ⟨?_, ?_⟩⟩⟩⟩
       · apply NotOccur.add
        · apply NotOccur.mul
         · exact NotOccur.var 8 hv8
         · apply NotOccur.succ; apply NotOccur.mul
          · apply NotOccur.succ; exact NotOccur.var 2 hv2
          · exact NotOccur.var 1 hv1
        · exact NotOccur.var 3 hv3
       · apply NotOccur.succ; apply NotOccur.mul
        · apply NotOccur.succ; exact NotOccur.var 2 hv2
        · exact NotOccur.var 1 hv1
       · apply NotOccur.add
        · exact NotOccur.var 3 hv3
        · apply NotOccur.succ; exact NotOccur.var 9 hv9
     · exact NotOccur_of_isClosed _ (isClosed_natToTerm A) v
    · exact NotOccur_of_isClosed _ (isClosed_natToTerm B) v
  · exact NotOccur_of_isClosed _ (isClosed_natToTerm k) v
 · exact NotOccur_of_isClosed _ (isClosed_natToTerm vk) v

lemma NotFreeForm_C5_closed {s : Nat} (A B k vk1 : Nat) (v : Nat)
   (hv0 : v ≠ 0) (hv1 : v ≠ 1) (hv2 : v ≠ 2) (hv3 : v ≠ 3) (hv8 : v ≠ 8) (hv9 : v ≠ 9) :
   NotFreeForm v (substForm 3 (natToTerm (s := s) vk1)
    (substForm 2 (natToTerm (s := s) (k + 1))
     (substForm 1 (natToTerm (s := s) B)
      (substForm 0 (natToTerm (s := s) A) BetaForm_sh)))) := by
```

```
  apply NotFreeForm_of_NotOccurForm
  apply NotOccurForm_subst_other
  · apply NotOccurForm_subst_other
    · apply NotOccurForm_subst_other
      · apply NotOccurForm_subst_other
        · dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form,
NotOccurForm]
          refine ⟨hv8, ⟨⟨NotOccur.var 0 hv0, ?_⟩, ⟨hv9, ⟨?_, ?_⟩⟩⟩⟩
          · apply NotOccur.add
            · apply NotOccur.mul
              · exact NotOccur.var 8 hv8
              · apply NotOccur.succ; apply NotOccur.mul
                · apply NotOccur.succ; exact NotOccur.var 2 hv2
                · exact NotOccur.var 1 hv1
            · exact NotOccur.var 3 hv3
          · apply NotOccur.succ; apply NotOccur.mul
            · apply NotOccur.succ; exact NotOccur.var 2 hv2
            · exact NotOccur.var 1 hv1
          · apply NotOccur.add
            · exact NotOccur.var 3 hv3
            · apply NotOccur.succ; exact NotOccur.var 9 hv9
        · exact NotOccur_of_isClosed _ (isClosed_natToTerm A) v
      · exact NotOccur_of_isClosed _ (isClosed_natToTerm B) v
    · exact NotOccur_of_isClosed _ (isClosed_natToTerm (k + 1)) v
  · exact NotOccur_of_isClosed _ (isClosed_natToTerm vk1) v


lemma NotFreeForm_C5_var6 {s : Nat} (A B k : Nat) (v : Nat)
    (hv0 : v ≠ 0) (hv1 : v ≠ 1) (hv2 : v ≠ 2) (hv3 : v ≠ 3) (hv6 : v ≠ 6) (hv8 : v ≠ 8) (hv9 : v
≠ 9) :
```

```
  NotFreeForm v (substForm 3 (ArithTerm.var (s := s) 6)
    (substForm 2 (ArithTerm.succ (natToTerm (s := s) k))
      (substForm 1 (natToTerm (s := s) B)
        (substForm 0 (natToTerm (s := s) A) BetaForm_sh)))) := by
 apply NotFreeForm_of_NotOccurForm
 apply NotOccurForm_subst_other
 · apply NotOccurForm_subst_other
  · apply NotOccurForm_subst_other
   · apply NotOccurForm_subst_other
    · dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, NotOccurForm]
      refine ⟨hv8, ⟨⟨NotOccur.var 0 hv0, ?_⟩, ⟨hv9, ⟨?_, ?_⟩⟩⟩⟩
      · apply NotOccur.add
       · apply NotOccur.mul
        · exact NotOccur.var 8 hv8
        · apply NotOccur.succ; apply NotOccur.mul
         · apply NotOccur.succ; exact NotOccur.var 2 hv2
         · exact NotOccur.var 1 hv1
       · exact NotOccur.var 3 hv3
      · apply NotOccur.succ; apply NotOccur.mul
       · apply NotOccur.succ; exact NotOccur.var 2 hv2
       · exact NotOccur.var 1 hv1
      · apply NotOccur.add
       · exact NotOccur.var 3 hv3
       · apply NotOccur.succ; exact NotOccur.var 9 hv9
    · exact NotOccur_of_isClosed _ (isClosed_natToTerm A) v
   · exact NotOccur_of_isClosed _ (isClosed_natToTerm B) v
  · apply NotOccur.succ
```

```
    exact NotOccur_of_isClosed _ (isClosed_natToTerm k) v
  · exact NotOccur.var 6 hv6
```

Lema `num_representable_graph`: Condición de gráfico para `NumForm`

Se demuestra la primera mitad de la representabilidad de `NumForm`: para cualquier número natural n, la instancia de `NumForm` con los valores correctos pertenece a Δ:

$$ \forall n \in \mathbb{N},\ \Delta \vdash \text{NumForm}(\overline{n}, \overline{\text{seq_code}(n)}) $$

Estructura de la demostración

La prueba es la más extensa y compleja del desarrollo hasta ahora, combinando todos los ingredientes previos. Se estructura en varias capas:

1. Obtención de parámetros beta: Por el teorema `exists_beta_seq` aplicado a la función $c(i) = \text{seq_code}(i)$, existen $A, B \in \mathbb{N}$ tales que $\text{godelBeta}(A, B, i) = \text{seq_code}(i)$ para todo $i \leq n$. En particular, para $i = 0$ y $i = n$.

2. Simplificación de `NumForm` instanciada: Se aplica `NumForm_subst_closed` con $w = \overline{n}$ y $z = \overline{\text{seq_code}(n)}$. La fórmula resultante es una conjunción de tres condiciones: `C1` (caso base), `C2` (elemento final), y `C3` (paso inductivo).

3. Verificación de `C1` y `C2`: Por `beta_sh_representable` con los parámetros A, B y los valores correspondientes (0, `zeroCode` para `C1`; n, `seq_code n` para `C2`), ambas subfórmulas pertenecen a Δ.

4. Verificación de `C3` (la condición inductiva): Esta es la parte más elaborada. Se debe demostrar que para todo k, si $k < \overline{n}$ entonces existen v, v' tales que $\beta(A, B, k) = v$, $\beta(A, B, k+1) = v'$, y $\text{IsSuccCode}(v, v')$.

La demostración procede por contradicción usando maximalidad y Henkin:

- Se asume que $\forall k\ \varphi(k) \notin \Delta$, donde $\varphi(k)$ es la fórmula del paso inductivo.

- Por maximalidad, $\neg \forall k\ \varphi(k) \in \Delta$, y por Henkin existe un testigo K (término cerrado) tal que $\neg \varphi(K) \in \Delta$.

- De $\neg \varphi(K)$ se deducen dos cosas:

  - $K < \overline{n} \in \Delta$ (la condición de orden se cumple).

  - $\neg \exists v, v'\ (\ldots) \in \Delta$ (no existen los testigos requeridos).

- De $K < \overline{n}$, por `lt_numeral_bounded` existe un $k < n$ tal que $K = \overline{k} \in \Delta$.

- Para ese k, se construyen explícitamente los testigos $v = \text{seq_code}(k)$ y $v' = \text{seq_code}(k+1)$. Por `beta_sh_representable`, se tiene $\beta(A, B, k) = \overline{v_k} \in \Delta$ y $\beta(A, B, k+1) = \overline{v_{k+1}} \in \Delta$. Por

`IsSuccCode_representable`, $\text{IsSuccCode}(\overline{v_k}, \overline{v_{k+1}}) \in \Delta$.

- Se sustituye K por $\overline{k}$ en la fórmula $\varphi(K)$ usando la igualdad $K = \overline{k}$ y la regla `eq_subst`. El resultado es $\varphi(\overline{k}) \in \Delta$, lo cual contradice $\neg \exists v, v'\ (\ldots) \in \Delta$.

5. Reconstrucción de los cuantificadores existenciales: Una vez verificadas `C1`, `C2` y `C3`, se construye la fórmula completa introduciendo secuencialmente los cuantificadores existenciales para v_{13} (parámetro B) y v_{12} (parámetro A). Esto requiere simplificar las sustituciones de v_{12} y v_{13} en las subfórmulas, utilizando extensivamente `substTerm_fresh` para eliminar las sustituciones sobre numerales cerrados.

Significado

`num_representable_graph` es uno de los resultados cumbre del desarrollo. Establece que la construcción inductiva de los numerales —desde `zero` hasta $\overline{n}$, pasando por todos los sucesores intermedios— puede ser capturada internamente por la fórmula `NumForm`. La demostración combina:

- El teorema de existencia del metalenguaje (`exists_beta_seq`) para obtener parámetros A, B.

- Los resultados de representabilidad de `BetaForm_sh` e `IsSuccCode` para verificar cada paso de la construcción.

- La maquinaria de maximalidad, Henkin, y `lt_numeral_bounded` para reducir cuantificadores universales sobre todos los términos cerrados a cuantificación sobre numerales.

- Extensas simplificaciones de sustituciones para manejar la complejidad sintáctica de `NumForm`.

Este lema establece que `NumForm` satisface la condición de gráfico de `IsRepresentable` para la función $n \mapsto \text{seq_code}(n)$. La condición de funcionalidad (unicidad) se demostrará en un lema complementario.

--
==

-- PASO 3: DEMOSTRACIÓN PRINCIPAL DE NUM_REPRESENTABLE_GRAPH

--
==

lemma num_representable_graph {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

(h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

```
  (n : Nat) :

  substForm 1 (natToTerm (s := s) (seq_code n)) (substForm 0 (natToTerm (s := s) n) NumForm) ∈ Δ := by

 let c_seq : Nat → Nat := fun i => seq_code i

 obtain ⟨A, B, h_beta⟩ := exists_beta_seq c_seq n

 have h_b0 : godelBeta A B 0 = zeroCode := by

  have h := h_beta 0 (by omega); dsimp [c_seq, seq_code] at h; exact h

 have h_bn : godelBeta A B n = seq_code n := by

  have h := h_beta n (by omega); dsimp [c_seq] at h; exact h


 rw [NumForm_subst_closed (natToTerm n) (natToTerm (seq_code n)) (isClosed_natToTerm n) (isClosed_natToTerm (seq_code n))]

 have hC1 := (beta_sh_representable Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ A B 0 zeroCode h_b0.symm).1

 have hC2 := (beta_sh_representable Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ A B n (seq_code n) h_bn.symm).1


 let φ_body : ArithForm s := ArithForm.impl (lt_form (ArithTerm.var (s := s) 4) (natToTerm (s := s) n) 11)

  (exists_form 5 (exists_form 6 (and_form

   (substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (ArithTerm.var (s := s) 4) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

    (and_form

     (substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4)) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

      IsSuccCode))))


 have hC3 : ArithForm.forall_form 4 φ_body ∈ Δ := by

  by_contra h_not

  have h_neg : ArithForm.neg (ArithForm.forall_form 4 φ_body) ∈ Δ := by
```

```
    cases h_max (ArithForm.forall_form 4 φ_body) with | inl h => contradiction | inr h
=> exact h

  have h_nd : ArithForm.neg (ArithForm.forall_form 4 (ArithForm.neg (ArithForm.neg
φ_body))) ∈ Δ := by

    rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con 4]

  obtain ⟨K, hK⟩ := h_henkin 4 _ h_nd

  dsimp [φ_body, substForm] at hK

  have hK_ded := SPA_Deduction_Internal.hyp Δ _ hK

  have h_lt_K_ded := imp_neg_elim_left Δ _ _ hK_ded

  have h_neg_ex_ded := imp_neg_elim_right Δ _ _ hK_ded

  have h_lt_K := maximal_consistent_closed_under_deduction Δ h_max h_con _
h_lt_K_ded

  have h_neg_ex := maximal_consistent_closed_under_deduction Δ h_max h_con _
h_neg_ex_ded


  have h_lt_01 : substForm 0 K.val (lt_form (ArithTerm.var 0) (natToTerm n) 1) ∈ Δ :=
by

    have h_lt_K_simp : substForm 4 K.val (lt_form (ArithTerm.var 4) (natToTerm n)
11) =

                exists_form 11 (ArithForm.eq (natToTerm n) (ArithTerm.add K.val
(ArithTerm.succ (ArithTerm.var 11)))) := by

     dsimp [lt_form, exists_form, ArithForm.exists_form, substForm, substTerm]

     rw [substTerm_fresh (natToTerm n) 4 K.val (NotOccur_of_isClosed _
(isClosed_natToTerm n) 4)]

    have h_lt_K_in : exists_form 11 (ArithForm.eq (natToTerm n) (ArithTerm.add K.val
(ArithTerm.succ (ArithTerm.var 11)))) ∈ Δ := by

     rwa [h_lt_K_simp] at h_lt_K

    obtain ⟨d11, hd11⟩ := h_henkin 11 _ h_lt_K_in

  have hd11_simp : substForm 11 d11.val (ArithForm.eq (natToTerm n)
(ArithTerm.add K.val (ArithTerm.succ (ArithTerm.var 11)))) =

            ArithForm.eq (natToTerm n) (ArithTerm.add K.val (ArithTerm.succ
d11.val)) := by
```

dsimp [substForm, substTerm]

rw [substTerm_fresh (natToTerm n) 11 d11.val (NotOccur_of_isClosed _ (isClosed_natToTerm n) 11),

substTerm_fresh K.val 11 d11.val (NotOccur_of_isClosed _ K.property 11)]

rw [hd11_simp] at hd11

have h_subst1 : substForm 1 d11.val (ArithForm.eq (natToTerm n) (ArithTerm.add K.val (ArithTerm.succ (ArithTerm.var 1)))) =

ArithForm.eq (natToTerm n) (ArithTerm.add K.val (ArithTerm.succ d11.val)) := by

dsimp [substForm, substTerm]

rw [substTerm_fresh (natToTerm n) 1 d11.val (NotOccur_of_isClosed _ (isClosed_natToTerm n) 1),

substTerm_fresh K.val 1 d11.val (NotOccur_of_isClosed _ K.property 1)]

have h_subst1_in : substForm 1 d11.val (ArithForm.eq (natToTerm n) (ArithTerm.add K.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ := by

rwa [h_subst1]

have hfree1 : IsFreeFor 1 d11.val (ArithForm.eq (natToTerm n) (ArithTerm.add K.val (ArithTerm.succ (ArithTerm.var 1)))) := IsFreeFor_closed _ 1 d11

have hex1 := exists_intro Δ 1 _ d11.val hfree1 (SPA_Deduction_Internal.hyp Δ _ h_subst1_in)

have hex1_in : exists_form 1 (ArithForm.eq (natToTerm n) (ArithTerm.add K.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ :=

maximal_consistent_closed_under_deduction Δ h_max h_con _ hex1

have h_target_eq : substForm 0 K.val (lt_form (ArithTerm.var 0) (natToTerm n) 1) =

exists_form 1 (ArithForm.eq (natToTerm n) (ArithTerm.add K.val (ArithTerm.succ (ArithTerm.var 1)))) := by

dsimp [lt_form, exists_form, ArithForm.exists_form, substForm, substTerm]

rw [substTerm_fresh (natToTerm n) 0 K.val (NotOccur_of_isClosed _ (isClosed_natToTerm n) 0)]

rw [h_target_eq]

exact hex1_in

```
  obtain ⟨k, hk_lt, hk_eq⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ n K h_lt_01


  let vk := seq_code k

  let vk1 := seq_code (k + 1)

  have h_vk1_def : vk1 = pair 4 vk + 1 := rfl


  have h_bk := h_beta k (by omega); dsimp [c_seq] at h_bk

  have h_bk1 := h_beta (k + 1) (by omega); dsimp [c_seq] at h_bk1


  have hC4_k := (beta_sh_representable Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ A B k vk h_bk.symm).1

  have hC5_k1 := (beta_sh_representable Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ A B (k + 1) vk1 h_bk1.symm).1

  have hC6_k := (IsSuccCode_representable Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ vk vk1 h_vk1_def).1


  have h_and_inner := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ hC5_k1) (SPA_Deduction_Internal.hyp Δ _ hC6_k)

  have h_and_inner_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and_inner

  have h_and_all := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ hC4_k) (SPA_Deduction_Internal.hyp Δ _ h_and_inner_in)

  have h_and_all_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_and_all


  -- ★ body6 usa ArithTerm.succ (natToTerm k) para coincidir con body5

  let body6 : ArithForm s := and_form

   (substForm 3 (natToTerm vk) (substForm 2 (natToTerm k) (substForm 1 (natToTerm B) (substForm 0 (natToTerm A) BetaForm_sh))))

   (and_form
```

(substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (natToTerm k)) (substForm 1 (natToTerm B) (substForm 0 (natToTerm A) BetaForm_sh))))

(substForm 5 (natToTerm (s := s) vk) IsSuccCode))

-- ★ h_occ6_AB con (s := s)

have h_occ6_AB : NotOccurForm 6 (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh)) :=

NotOccurForm_Beta_subst_AB A B 6 (by decide) (by decide) (by decide) (by decide) (by decide) (by decide)

have h_occ6_X : NotOccurForm 6 (substForm 2 (ArithTerm.succ (natToTerm k)) (substForm 1 (natToTerm B) (substForm 0 (natToTerm A) BetaForm_sh))) :=

NotOccurForm_subst_other _ 2 6 (ArithTerm.succ (natToTerm k)) h_occ6_AB (NotOccur.succ _ (NotOccur_of_isClosed _ (isClosed_natToTerm k) 6))

-- ★ h_subst6: resultado esperado con succ(natToTerm k), sin (by decide : 5 ≠ 6)

have h_subst6 : substForm 6 (natToTerm (s := s) vk1) body6 =

and_form

(substForm 3 (natToTerm (s := s) vk) (substForm 2 (natToTerm (s := s) k) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(and_form

(substForm 3 (natToTerm (s := s) vk1) (substForm 2 (ArithTerm.succ (natToTerm (s := s) k)) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(substForm 6 (natToTerm (s := s) vk1) (substForm 5 (natToTerm (s := s) vk) IsSuccCode))) := by

dsimp [body6, and_form, substForm, substTerm]

rw [substForm_fresh_of_NotFreeForm _ 6 (natToTerm vk1) (NotFreeForm_C4_closed A B k vk 6 (by decide) (by decide) (by decide) (by decide) (by decide) (by decide))]

rw [substForm_substForm_term_comm _ 3 6 (natToTerm vk1) h_occ6_X]

have h_and_all_rew : substForm 6 (natToTerm vk1) body6 ∈ Δ := by rwa [h_subst6]

have hfree6 : IsFreeFor 6 (natToTerm vk1) body6 := IsFreeFor_closed _ 6 ⟨natToTerm vk1, isClosed_natToTerm vk1⟩

have hex6 := exists_intro Δ 6 body6 (natToTerm vk1) hfree6 (SPA_Deduction_Internal.hyp Δ _ h_and_all_rew)

have hex6_in : exists_form 6 body6 ∈ Δ := maximal_consistent_closed_under_deduction Δ h_max h_con _ hex6

let body5 : ArithForm s := exists_form 6 (and_form

(substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (natToTerm k) (substForm 1 (natToTerm B) (substForm 0 (natToTerm A) BetaForm_sh))))

(and_form

(substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (natToTerm k)) (substForm 1 (natToTerm B) (substForm 0 (natToTerm A) BetaForm_sh))))

IsSuccCode))

have h_occ5_AB : NotOccurForm 5 (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh)) :=

NotOccurForm_Beta_subst_AB A B 5 (by decide) (by decide) (by decide) (by decide) (by decide) (by decide)

have h_occ5_C4 : NotOccurForm 5 (substForm 2 (natToTerm (s := s) k) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) :=

NotOccurForm_subst_other _ 2 5 (natToTerm k) h_occ5_AB (NotOccur_of_isClosed _ (isClosed_natToTerm k) 5)

have h_subst5 : substForm 5 (natToTerm (s := s) vk) body5 = exists_form 6 body6 := by

dsimp [body5, body6, exists_form, ArithForm.exists_form, and_form, substForm, substTerm]

rw [substForm_substForm_term_comm _ 3 5 (natToTerm vk) h_occ5_C4]

rw [substForm_fresh_of_NotFreeForm _ 5 (natToTerm vk) (NotFreeForm_C5_var6 A B k 5 (by decide) (by decide) (by decide) (by decide) (by decide) (by decide) (by decide))]

have hex6_rew : substForm 5 (natToTerm vk) body5 ∈ Δ := by rwa [h_subst5]

have hfree5 : IsFreeFor 5 (natToTerm vk) body5 := IsFreeFor_closed _ 5 ⟨natToTerm vk, isClosed_natToTerm vk⟩

have hex5 := exists_intro Δ 5 body5 (natToTerm vk) hfree5 (SPA_Deduction_Internal.hyp Δ _ hex6_rew)

have hex5_in : exists_form 5 body5 ∈ Δ := maximal_consistent_closed_under_deduction Δ h_max h_con _ hex5

have h_k_K : ArithForm.eq (natToTerm k) K.val ∈ Δ := eq_symm_Δ h_max h_con hk_eq

let φ_K : ArithForm s := exists_form 5 (exists_form 6 (and_form

(substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (ArithTerm.var (s := s) 4) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(and_form

(substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4)) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

IsSuccCode)))

have hfree_k : IsFreeFor 4 (natToTerm k) φ_K := IsFreeFor_closed _ 4 ⟨natToTerm k, isClosed_natToTerm k⟩

have hfree_K : IsFreeFor 4 K.val φ_K := IsFreeFor_closed _ 4 K

have h_occ4_AB : NotOccurForm 4 (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh)) :=

NotOccurForm_Beta_subst_AB A B 4 (by decide) (by decide) (by decide) (by decide) (by decide) (by decide)

-- ★ h_subst_k: h_comm43_2 con dsimp [BetaForm_sh,...] + repeat rw

have h_subst_k : substForm 4 (natToTerm k) φ_K = exists_form 5 body5 := by

dsimp [φ_K, body5, exists_form, ArithForm.exists_form, and_form, substForm, substTerm]

-- Primera conjunción: conmutar 4 con 3, luego 4 con 2

have h_comm43_1 := substForm_comm (substForm 2 (ArithTerm.var (s := s) 4) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) 4 3 (natToTerm (s := s) k) (ArithTerm.var 5) (by decide) (NotOccur_of_isClosed _ (isClosed_natToTerm k) 3) (NotOccur.var 5 (by decide : 4 ≠ 5))

dsimp [substTerm] at h_comm43_1

rw [substForm_substForm_term_comm _ 2 4 (natToTerm k) h_occ4_AB] at h_comm43_1

rw [h_comm43_1]

-- Segunda conjunción: conmutar 4 con 3, luego evaluar 4 sobre succ(var 4)

have h_comm43_2 : substForm 4 (natToTerm k)

(substForm 3 (ArithTerm.var (s := s) 6)

(substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh)))) =

substForm 3 (ArithTerm.var (s := s) 6)

(substForm 2 (ArithTerm.succ (natToTerm (s := s) k))

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) := by

rw [substForm_comm _ 4 3 (natToTerm k) (ArithTerm.var 6) (by decide) (NotOccur_of_isClosed _ (isClosed_natToTerm k) 3) (NotOccur.var 6 (by decide : 4 ≠ 6))]

dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, substForm, substTerm]

rw [substTerm_fresh (natToTerm A) 1 (natToTerm B) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 1)]

rw [substTerm_fresh (natToTerm A) 2 (ArithTerm.var 4).succ (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

rw [substTerm_fresh (natToTerm A) 4 (natToTerm k) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 4)]

rw [substTerm_fresh (natToTerm A) 2 (natToTerm k).succ (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

rw [substTerm_fresh (natToTerm B) 2 (ArithTerm.var 4).succ (NotOccur_of_isClosed _ (isClosed_natToTerm B) 2)]

rw [substTerm_fresh (natToTerm B) 4 (natToTerm k) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 4)]

rw [substTerm_fresh (natToTerm B) 2 (natToTerm k).succ (NotOccur_of_isClosed _ (isClosed_natToTerm B) 2)]

rw [h_comm43_2]

rw [substForm_fresh_of_NotFreeForm IsSuccCode 4 (natToTerm k) (NotFreeForm_IsSuccCode 4 (by decide) (by decide))]

have hex5_in_subst : substForm 4 (natToTerm k) φ_K ∈ Δ := by rwa [h_subst_k]

have h_subst_K_ded := SPA_Deduction_Internal.eq_subst Δ 4 φ_K (natToTerm k) K.val

(SPA_Deduction_Internal.hyp Δ _ h_k_K) (SPA_Deduction_Internal.hyp Δ _ hex5_in_subst) hfree_k hfree_K

have h_subst_K_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_subst_K_ded

have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=

SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_subst_K_in) (SPA_Deduction_Internal.hyp Δ _ h_neg_ex)

exact h_con h_bot

have hC23 := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ hC2) (SPA_Deduction_Internal.hyp Δ _ hC3)

have hC23_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ hC23

have h_all := and_intro Δ _ _ (SPA_Deduction_Internal.hyp Δ _ hC1) (SPA_Deduction_Internal.hyp Δ _ hC23_in)

have h_all_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_all

let body3 : ArithForm s := and_form

(substForm 3 (natToTerm (s := s) zeroCode) (substForm 2 (natToTerm (s := s) 0) (substForm 1 (ArithTerm.var (s := s) 13) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(and_form

(substForm 3 (natToTerm (s := s) (seq_code n)) (substForm 2 (natToTerm (s := s) n) (substForm 1 (ArithTerm.var (s := s) 13) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(ArithForm.forall_form 4 (ArithForm.impl (lt_form (ArithTerm.var (s := s) 4) (natToTerm (s := s) n) 11)

(exists_form 5 (exists_form 6 (and_form

(substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (ArithTerm.var (s := s) 4) (substForm 1 (ArithTerm.var (s := s) 13) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(and_form

(substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4)) (substForm 1 (ArithTerm.var (s := s) 13) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

IsSuccCode)))))))

-- h_subst3: repeat rw con natToTerm B como término de sustitución de var 1

have h_subst3 : substForm 13 (natToTerm (s := s) B) body3 =

and_form

(substForm 3 (natToTerm (s := s) zeroCode) (substForm 2 (natToTerm (s := s) 0) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(and_form

(substForm 3 (natToTerm (s := s) (seq_code n)) (substForm 2 (natToTerm (s := s) n) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(ArithForm.forall_form 4 (ArithForm.impl (lt_form (ArithTerm.var (s := s) 4) (natToTerm (s := s) n) 11)

(exists_form 5 (exists_form 6 (and_form

(substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (ArithTerm.var (s := s) 4) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(and_form

(substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4)) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

IsSuccCode))))))) := by

dsimp [body3, BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, substForm, substTerm, natToTerm]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 1 (natToTerm (s := s) B) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 1)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 2 ArithTerm.zero (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) B) 2 ArithTerm.zero (NotOccur_of_isClosed _ (isClosed_natToTerm B) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 1 (ArithTerm.var 13) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 1)]

repeat rw [substTerm_fresh (natToTerm (s := s) zeroCode) 13 (natToTerm (s := s) B) (NotOccur_of_isClosed _ (isClosed_natToTerm zeroCode) 13)]

repeat rw [substTerm_fresh (natToTerm (s := s) (seq_code n)) 13 (natToTerm (s := s) B) (NotOccur_of_isClosed _ (isClosed_natToTerm (seq_code n)) 13)]

repeat rw [substTerm_fresh (natToTerm (s := s) n) 13 (natToTerm (s := s) B) (NotOccur_of_isClosed _ (isClosed_natToTerm n) 13)]

rw [substForm_fresh_of_NotFreeForm IsSuccCode 13 (natToTerm (s := s) B) (NotFreeForm_IsSuccCode 13 (by decide) (by decide))]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 2 ArithTerm.zero (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 2 (natToTerm (s := s) n) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 2 (ArithTerm.var 4) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 2 (ArithTerm.var 4).succ (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 3 (natToTerm (s := s) zeroCode) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 3 (natToTerm (s := s) (seq_code n)) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 3 (ArithTerm.var 5) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 3 (ArithTerm.var 6) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 13 (natToTerm (s := s) B) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 13)]

repeat rw [substTerm_fresh (natToTerm (s := s) B) 2 (natToTerm (s := s) n) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) B) 2 (ArithTerm.var 4) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) B) 2 (ArithTerm.var 4).succ (NotOccur_of_isClosed _ (isClosed_natToTerm B) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) B) 3 (natToTerm (s := s) zeroCode) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) B) 3 (natToTerm (s := s) (seq_code n)) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) B) 3 (ArithTerm.var 5) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) B) 3 (ArithTerm.var 6) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) n) 3 (natToTerm (s := s) (seq_code n)) (NotOccur_of_isClosed _ (isClosed_natToTerm n) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) n) 13 (natToTerm (s := s) B) (NotOccur_of_isClosed _ (isClosed_natToTerm n) 13)]

have h_all_in_rew : substForm 13 (natToTerm B) body3 ∈ Δ := by rwa [h_subst3]

have hfree3 : IsFreeFor 13 (natToTerm B) body3 := IsFreeFor_closed _ 13 ⟨natToTerm B, isClosed_natToTerm B⟩

have hex3 := exists_intro Δ 13 body3 (natToTerm B) hfree3 (SPA_Deduction_Internal.hyp Δ _ h_all_in_rew)

have hex3_in : exists_form 13 body3 ∈ Δ := maximal_consistent_closed_under_deduction Δ h_max h_con _ hex3

let body2 : ArithForm s := exists_form 13 (and_form

(substForm 3 (natToTerm (s := s) zeroCode) (substForm 2 (natToTerm (s := s) 0) (substForm 1 (ArithTerm.var (s := s) 13) (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))

(and_form

(substForm 3 (natToTerm (s := s) (seq_code n)) (substForm 2 (natToTerm (s := s) n) (substForm 1 (ArithTerm.var (s := s) 13) (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))

(ArithForm.forall_form 4 (ArithForm.impl (lt_form (ArithTerm.var (s := s) 4) (natToTerm (s := s) n) 11)

(exists_form 5 (exists_form 6 (and_form

(substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (ArithTerm.var (s := s) 4) (substForm 1 (ArithTerm.var (s := s) 13) (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))

(and_form

(substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4)) (substForm 1 (ArithTerm.var (s := s) 13) (substForm 0 (ArithTerm.var (s := s) 12) BetaForm_sh))))

IsSuccCode))))))))

-- ★ h_subst2: análogo con var 2 y natToTerm A

have h_subst2 : substForm 12 (natToTerm (s := s) A) body2 = exists_form 13 body3 := by

dsimp [body2, body3, BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, substForm, substTerm, natToTerm]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 1 (ArithTerm.var 13) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 1)]

repeat rw [substTerm_fresh (natToTerm (s := s) zeroCode) 12 (natToTerm (s := s) A) (NotOccur_of_isClosed _ (isClosed_natToTerm zeroCode) 12)]

repeat rw [substTerm_fresh (natToTerm (s := s) n) 12 (natToTerm (s := s) A) (NotOccur_of_isClosed _ (isClosed_natToTerm n) 12)]

repeat rw [substTerm_fresh (natToTerm (s := s) (seq_code n)) 12 (natToTerm (s := s) A) (NotOccur_of_isClosed _ (isClosed_natToTerm (seq_code n)) 12)]

rw [substForm_fresh_of_NotFreeForm IsSuccCode 12 (natToTerm (s := s) A) (NotFreeForm_IsSuccCode 12 (by decide) (by decide))]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 2 ArithTerm.zero (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 2 (natToTerm (s := s) n) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 2 (ArithTerm.var 4) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 2 (ArithTerm.var 4).succ (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 3 (natToTerm (s := s) zeroCode) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 3 (natToTerm (s := s) (seq_code n)) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 3 (ArithTerm.var 5) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) A) 3 (ArithTerm.var 6) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) n) 3 (natToTerm (s := s) (seq_code n)) (NotOccur_of_isClosed _ (isClosed_natToTerm n) 3)]

repeat rw [substTerm_fresh (natToTerm (s := s) n) 12 (natToTerm (s := s) A) (NotOccur_of_isClosed _ (isClosed_natToTerm n) 12)]

have hex3_in_rew : substForm 12 (natToTerm A) body2 ∈ Δ := by rwa [h_subst2]

have hfree2 : IsFreeFor 12 (natToTerm A) body2 := IsFreeFor_closed _ 12 ⟨natToTerm A, isClosed_natToTerm A⟩

have hex2 := exists_intro Δ 12 body2 (natToTerm A) hfree2 (SPA_Deduction_Internal.hyp Δ _ hex3_in_rew)

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ hex2

Lema `add_cancel_right_Δ`: Cancelación aditiva por la derecha en Δ

Se demuestra una propiedad algebraica fundamental de la adición dentro de Δ: la cancelación por la derecha. Si dos términos cerrados a y b satisfacen $a + c = b + c$ para algún término cerrado c, entonces $a = b$:

$$ \forall a, b, c \in \text{TermClosed } s,\ \Delta \vdash a + c = b + c \implies \Delta \vdash a = b $$

Lema auxiliar `substForm_phi_prime_cancel`

Define una fórmula de inducción $\varphi'(v)$ específica para la cancelación y demuestra cómo se comporta bajo sustitución. La fórmula es:

$$ \varphi'(v) := \text{Ax0}_a \to (\text{Ax0}_b \to (\forall v\ \text{Ax1}_a(v)) \to (\forall v\ \text{Ax1}_b(v)) \to (a + v = b + v \to a = b)) $$

El lema establece que $\text{substForm}(v, u, \varphi')$ solo afecta a la igualdad $a + v = b + v$, reemplazando v por u y dejando el resto de la fórmula inalterado. Esto es posible porque a y b son términos cerrados (no contienen a v), y los cuantificadores $\forall v$ ligan la variable v, por lo que la sustitución no los atraviesa.

Estructura de la demostración de `add_cancel_right_Δ`

La prueba sigue el patrón de inducción interna sobre la variable v, utilizando el tercer término c como valor a instanciar.

1. Variable fresca y construcción de φ': Se elige v fresca respecto a a y b. La fórmula $\varphi'(v)$ codifica el paso inductivo de la cancelación.

2. Caso base ($v := 0$): Se debe probar que $a + 0 = b + 0 \to a = b$ bajo los axiomas. Usando Ax0_a y Ax0_b (los axiomas de cero por la derecha) y la hipótesis $a + 0 = b + 0$, se obtiene $a = b$ por transitividad.

3. Paso inductivo ($v \to S(v)$): Se asume $\varphi'(v)$ (la hipótesis de inducción) y se debe probar $\varphi'(S(v))$. Bajo la hipótesis adicional $a + S(v) = b + S(v)$, se tiene:

- Por $\text{Ax1}_a(v)$: $a + S(v) = S(a + v)$

- Por la igualdad asumida y $\text{Ax1}_b(v)$: $S(a + v) = b + S(v) = S(b + v)$

- Por el segundo axioma de Peano (inyectividad del sucesor): $S(a + v) = S(b + v) \to a + v = b + v$

- Por la hipótesis de inducción $\varphi'(v)$: $a + v = b + v \to a = b$

- Por tanto, $a = b$.

4. Aplicación de la inducción: El esquema de inducción proporciona $\forall v\ \varphi'(v)$. Instanciando con c, se obtiene $\varphi'(c) \in \Delta$.

5. Descarga de axiomas: Las premisas de $\varphi'(c)$ se descargan:

- Ax0_a y Ax0_b están en Δ por `h_add_zero`.

- $\forall v\ \text{Ax1}_a(v)$ y $\forall v\ \text{Ax1}_b(v)$ se demuestra que pertenecen a Δ usando maximalidad y Henkin (si no pertenecieran, existiría un contraejemplo que contradice `h_add_succ`).

- La hipótesis $a + c = b + c$ es exactamente la premisa del lema.

Tras cinco eliminaciones de la implicación, se obtiene $a = b \in \Delta$.

Significado

`add_cancel_right_Δ` es una propiedad algebraica que, aunque elemental en el modelo estándar, requiere una demostración no trivial dentro del sistema formal. Es necesaria para establecer la inyectividad de ciertas operaciones en la aritmetización, en particular para demostrar la unicidad en la representabilidad de `NumForm`.

En la prueba de funcionalidad de `NumForm`, será necesario deducir igualdades entre códigos a partir de igualdades entre sumas que involucran esos códigos. La cancelación aditiva proporciona la herramienta para eliminar términos comunes y obtener las igualdades deseadas. La demostración, aunque extensa, sigue el patrón ya establecido de inducción interna con verificación de pertenencia de los axiomas cuantificados mediante maximalidad y Henkin, confirmando que este patrón es una herramienta sistemática y reutilizable para derivar propiedades algebraicas en Δ.

--
==========

-- LEMA DE CANCELACIÓN ADITIVA: a + c = b + c → a = b

--
==========

```
/--
  Lema auxiliar de sustitución uniforme para la fórmula inductiva de cancelación aditiva.
  Garantiza que la sustitución de la variable v por un término u solo afecta a los
  componentes donde v ocurre libre (la igualdad atómica Eq_v).
-/
lemma substForm_phi_prime_cancel {s : Nat} (a b : TermClosed s) (v : Nat)
  (hva : NotOccur v a.val) (hvb : NotOccur v b.val) (u : ArithTerm s) :
  let Ax0_a : ArithForm s := add_zero_axiom a.val
  let Ax0_b : ArithForm s := add_zero_axiom b.val
  let Ax1_a : ArithForm s := ArithForm.forall_form v (add_succ_axiom a.val (ArithTerm.var v))
  let Ax1_b : ArithForm s := ArithForm.forall_form v (add_succ_axiom b.val (ArithTerm.var v))
  let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.add a.val (ArithTerm.var v)) (ArithTerm.add b.val (ArithTerm.var v))
  let Goal : ArithForm s := ArithForm.eq a.val b.val
  let φ' : ArithForm s := ArithForm.impl Ax0_a (ArithForm.impl Ax0_b (ArithForm.impl Ax1_a (ArithForm.impl Ax1_b (ArithForm.impl Eq_v Goal))))
  substForm v u φ' =
  ArithForm.impl Ax0_a (ArithForm.impl Ax0_b (ArithForm.impl Ax1_a (ArithForm.impl Ax1_b (ArithForm.impl (ArithForm.eq (ArithTerm.add a.val u) (ArithTerm.add b.val u)) Goal)))) := by
 intro Ax0_a Ax0_b Ax1_a Ax1_b Eq_v Goal φ'
 dsimp [φ', Ax0_a, Ax0_b, Ax1_a, Ax1_b, Eq_v, Goal, add_zero_axiom, add_succ_axiom, substForm, substTerm]
 rw [substTerm_fresh a.val v u hva]
 rw [substTerm_fresh b.val v u hvb]
```

have h_vv : (if v = v then u else ArithTerm.var v) = u := if_pos rfl

have h_if1 : (if v = v then ArithForm.forall_form v (ArithForm.eq (ArithTerm.add a.val (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add a.val (ArithTerm.var v))))

else ArithForm.forall_form v (ArithForm.eq (ArithTerm.add a.val (ArithTerm.succ u)) (ArithTerm.succ (ArithTerm.add a.val u)))) =

ArithForm.forall_form v (ArithForm.eq (ArithTerm.add a.val (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add a.val (ArithTerm.var v)))) := if_pos rfl

have h_if2 : (if v = v then ArithForm.forall_form v (ArithForm.eq (ArithTerm.add b.val (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add b.val (ArithTerm.var v))))

else ArithForm.forall_form v (ArithForm.eq (ArithTerm.add b.val (ArithTerm.succ u)) (ArithTerm.succ (ArithTerm.add b.val u)))) =

ArithForm.forall_form v (ArithForm.eq (ArithTerm.add b.val (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add b.val (ArithTerm.var v)))) := if_pos rfl

rw [h_vv, h_if1, h_if2]

/--

Teorema de Cancelación por la Derecha para la Adición en Δ.

Demuestra internamente en el sistema deductivo objeto (vía inducción de Peano sobre v)

que para términos cerrados a, b, c, la igualdad a + c = b + c implica a = b.

-/

lemma add_cancel_right_Δ {s : Nat} (Δ : Set (ArithForm s))

(h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(a b c : TermClosed s)

(h : ArithForm.eq (ArithTerm.add a.val c.val) (ArithTerm.add b.val c.val) ∈ Δ) :

ArithForm.eq a.val b.val ∈ Δ := by

```
  let v := max_var_term a.val + max_var_term b.val + 1
  have hva : NotOccur v a.val := NotOccur_above_max_var_term a.val v (by omega)
  have hvb : NotOccur v b.val := NotOccur_above_max_var_term b.val v (by omega)

  let Ax0_a : ArithForm s := add_zero_axiom a.val
  let Ax0_b : ArithForm s := add_zero_axiom b.val
  let Ax1_a_body : ArithForm s := add_succ_axiom a.val (ArithTerm.var v)
  let Ax1_a : ArithForm s := ArithForm.forall_form v Ax1_a_body
  let Ax1_b_body : ArithForm s := add_succ_axiom b.val (ArithTerm.var v)
  let Ax1_b : ArithForm s := ArithForm.forall_form v Ax1_b_body

  let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.add a.val (ArithTerm.var v)) (ArithTerm.add b.val (ArithTerm.var v))
  let Goal : ArithForm s := ArithForm.eq a.val b.val

  let φ' : ArithForm s := ArithForm.impl Ax0_a (ArithForm.impl Ax0_b (ArithForm.impl Ax1_a (ArithForm.impl Ax1_b (ArithForm.impl Eq_v Goal))))

  have h_base : SPA_Deduction_Internal s ∅ (substForm v ArithTerm.zero φ') := by
    rw [substForm_phi_prime_cancel a b v hva hvb ArithTerm.zero]
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    let Γ5 : Set (ArithForm s) := insert (ArithForm.eq (ArithTerm.add a.val ArithTerm.zero) (ArithTerm.add b.val ArithTerm.zero))
      (insert Ax1_b (insert Ax1_a (insert Ax0_b (insert Ax0_a ∅))))
    have h_Ax0_a : SPA_Deduction_Internal s Γ5 Ax0_a := SPA_Deduction_Internal.hyp Γ5 Ax0_a (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))
```

have h_Ax0_b : SPA_Deduction_Internal s Γ5 Ax0_b := SPA_Deduction_Internal.hyp Γ5 Ax0_b (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

have h_Eq0 : SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.add a.val ArithTerm.zero) (ArithTerm.add b.val ArithTerm.zero)) :=

SPA_Deduction_Internal.hyp Γ5 _ (Or.inl rfl)

have h_symm_a := SPA_Deduction_Internal.eq_symm Γ5 _ _ h_Ax0_a

have h_trans1 := SPA_Deduction_Internal.eq_trans Γ5 _ _ _ h_symm_a h_Eq0

exact SPA_Deduction_Internal.eq_trans Γ5 _ _ _ h_trans1 h_Ax0_b

have h_step : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v (ArithForm.impl φ' (substForm v (ArithTerm.succ (ArithTerm.var v)) φ'))) := by

apply SPA_Deduction_Internal.forall_intro

· rw [substForm_phi_prime_cancel a b v hva hvb (ArithTerm.succ (ArithTerm.var v))]

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

let Γ6 : Set (ArithForm s) := insert (ArithForm.eq (ArithTerm.add a.val (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.add b.val (ArithTerm.succ (ArithTerm.var v))))

(insert Ax1_b (insert Ax1_a (insert Ax0_b (insert Ax0_a (insert φ' ∅)))))

have h_φ' : SPA_Deduction_Internal s Γ6 φ' := SPA_Deduction_Internal.hyp Γ6 φ' (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl))))))

have h_Ax0_a : SPA_Deduction_Internal s Γ6 Ax0_a := SPA_Deduction_Internal.hyp Γ6 Ax0_a (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))

have h_Ax0_b : SPA_Deduction_Internal s Γ6 Ax0_b := SPA_Deduction_Internal.hyp Γ6 Ax0_b (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

have h_Ax1_a : SPA_Deduction_Internal s Γ6 Ax1_a := SPA_Deduction_Internal.hyp Γ6 Ax1_a (Or.inr (Or.inr (Or.inl rfl)))

have h_Ax1_b : SPA_Deduction_Internal s Γ6 Ax1_b := SPA_Deduction_Internal.hyp Γ6 Ax1_b (Or.inr (Or.inl rfl))

have h_Eq_sv : SPA_Deduction_Internal s Γ6 (ArithForm.eq (ArithTerm.add a.val (ArithTerm.succ (ArithTerm.var v))) (ArithTerm.add b.val (ArithTerm.succ (ArithTerm.var v)))) :=

SPA_Deduction_Internal.hyp Γ6 _ (Or.inl rfl)

have h_imp1 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_φ' h_Ax0_a

have h_imp2 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp1 h_Ax0_b

have h_imp3 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp2 h_Ax1_a

have h_imp4 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp3 h_Ax1_b

have h_Ax1_a_v := SPA_Deduction_Internal.forall_elim Γ6 v Ax1_a_body (ArithTerm.var v) h_Ax1_a True.intro

change SPA_Deduction_Internal s Γ6 (ArithForm.eq (ArithTerm.add (substTerm v (ArithTerm.var v) a.val) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add (substTerm v (ArithTerm.var v) a.val) (substTerm v (ArithTerm.var v) (ArithTerm.var v))))) at h_Ax1_a_v

rw [substTerm_fresh a.val v (ArithTerm.var v) hva, substTerm_var_same v (ArithTerm.var v)] at h_Ax1_a_v

have h_Ax1_b_v := SPA_Deduction_Internal.forall_elim Γ6 v Ax1_b_body (ArithTerm.var v) h_Ax1_b True.intro

change SPA_Deduction_Internal s Γ6 (ArithForm.eq (ArithTerm.add (substTerm v (ArithTerm.var v) b.val) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add (substTerm v (ArithTerm.var v) b.val) (substTerm v (ArithTerm.var v) (ArithTerm.var v))))) at h_Ax1_b_v

rw [substTerm_fresh b.val v (ArithTerm.var v) hvb, substTerm_var_same v (ArithTerm.var v)] at h_Ax1_b_v

have h_symm_a := SPA_Deduction_Internal.eq_symm Γ6 _ _ h_Ax1_a_v

have h_t1 := SPA_Deduction_Internal.eq_trans Γ6 _ _ _ h_symm_a h_Eq_sv

have h_t2 := SPA_Deduction_Internal.eq_trans Γ6 _ _ _ h_t1 h_Ax1_b_v

```
    have h_peano2 : SPA_Deduction_Internal s Γ6 (ArithForm.impl (ArithForm.eq (ArithTerm.succ (ArithTerm.add a.val (ArithTerm.var v))) (ArithTerm.succ (ArithTerm.add b.val (ArithTerm.var v)))) (ArithForm.eq (ArithTerm.add a.val (ArithTerm.var v)) (ArithTerm.add b.val (ArithTerm.var v)))) :=

      SPA_Deduction_Internal.ax_peano2 Γ6 (ArithTerm.add a.val (ArithTerm.var v)) (ArithTerm.add b.val (ArithTerm.var v))


    have h_Eq_v := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_peano2 h_t2

    exact SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp4 h_Eq_v

  · intro ψ hψ

    nomatch hψ


 have h_ind : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ') :=

   SPA_Deduction_Internal.induction ∅ φ' v h_base h_step


 have h_ind_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form v φ') :=

   SPA_Deduction_Internal.weaken ∅ _ Δ h_ind (Set.empty_subset _)


 have h_free : IsFreeFor v c.val φ' := IsFreeFor_closed φ' v c

 have h_elim := SPA_Deduction_Internal.forall_elim Δ v φ' c.val h_ind_Δ h_free

 rw [substForm_phi_prime_cancel a b v hva hvb c.val] at h_elim


 have h_Ax0_a_Δ : SPA_Deduction_Internal s Δ Ax0_a := SPA_Deduction_Internal.hyp Δ _ (h_add_zero a.val)

 have h_Ax0_b_Δ : SPA_Deduction_Internal s Δ Ax0_b := SPA_Deduction_Internal.hyp Δ _ (h_add_zero b.val)


 have h_elim1 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim h_Ax0_a_Δ

 have h_elim2 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim1 h_Ax0_b_Δ
```

```
  have h_Ax1_a_in : Ax1_a ∈ Δ := by
    by_contra h_not
    have h_neg : ArithForm.neg Ax1_a ∈ Δ := by cases h_max Ax1_a with | inl h' => contradiction | inr h' => exact h'
    have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_a_body))) ∈ Δ := by
      rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
    obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_a_body) h_nd
    have h_subst_ax : substForm v t.val Ax1_a_body =
       ArithForm.eq (ArithTerm.add a.val (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add a.val t.val)) := by
      dsimp [Ax1_a_body, add_succ_axiom, substForm, substTerm]
      rw [substTerm_fresh a.val v t.val hva]
      have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
      rw [h_if]
    have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.add a.val (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add a.val t.val))) ∈ Δ := by
      change ArithForm.neg (substForm v t.val Ax1_a_body) ∈ Δ at ht
      rw [h_subst_ax] at ht
      exact ht
    have h_pos := h_add_succ a.val t.val
    exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))


  have h_Ax1_b_in : Ax1_b ∈ Δ := by
    by_contra h_not
    have h_neg : ArithForm.neg Ax1_b ∈ Δ := by cases h_max Ax1_b with | inl h' => contradiction | inr h' => exact h'
    have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_b_body))) ∈ Δ := by
```

```
    rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_b_body) h_nd
  have h_subst_ax : substForm v t.val Ax1_b_body =
    ArithForm.eq (ArithTerm.add b.val (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add b.val t.val)) := by
    dsimp [Ax1_b_body, add_succ_axiom, substForm, substTerm]
    rw [substTerm_fresh b.val v t.val hvb]
    have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
    rw [h_if]
  have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.add b.val (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add b.val t.val))) ∈ Δ := by
    change ArithForm.neg (substForm v t.val Ax1_b_body) ∈ Δ at ht
    rw [h_subst_ax] at ht
    exact ht
  have h_pos := h_add_succ b.val t.val
  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))

 have h_Ax1_a_Δ : SPA_Deduction_Internal s Δ Ax1_a := SPA_Deduction_Internal.hyp Δ _ h_Ax1_a_in
 have h_Ax1_b_Δ : SPA_Deduction_Internal s Δ Ax1_b := SPA_Deduction_Internal.hyp Δ _ h_Ax1_b_in
 have h_Eq_c_Δ : SPA_Deduction_Internal s Δ (ArithForm.eq (ArithTerm.add a.val c.val) (ArithTerm.add b.val c.val)) := SPA_Deduction_Internal.hyp Δ _ h

 have h_elim3 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim2 h_Ax1_a_Δ
 have h_elim4 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim3 h_Ax1_b_Δ
 have h_final_ded := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim4 h_Eq_c_Δ

 exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final_ded
```

Lema `add_assoc_Δ_term`: Asociatividad de la adición para términos genéricos en Δ

Se demuestra la propiedad asociativa de la adición para tres términos arbitrarios (no necesariamente cerrados) dentro de Δ. Esta versión generaliza el lema `add_assoc_Δ` (que requería que los tres términos fueran cerrados) a cualquier terna de términos:

$$ \forall a, b, c \in \text{ArithTerm } s,\ \Delta \vdash (a + b) + c = a + (b + c) $$

Lema auxiliar `substForm_phi_prime_assoc`

Define la fórmula de inducción para la asociatividad y demuestra su comportamiento bajo sustitución. La fórmula es:

$$ \varphi'(v) := \text{Ax0}_{a+b} \to (\text{Ax0}_b \to (\forall v\ \text{Ax1}_{a+b}(v)) \to (\forall v\ \text{Ax1}_b(v)) \to (\forall v\ \text{Ax1}_a(v)) \to (a + b) + v = a + (b + v)) $$

El lema establece que la sustitución de v por u solo afecta a la igualdad final, reemplazando v por u en los lugares donde aparece libre, y dejando inalterados los axiomas (que están fuera del alcance de v o tienen a v ligada). La demostración utiliza `substTerm_fresh` para a y b (pues v no ocurre en ellos por la elección de v como variable fresca).

Estructura de la demostración de `add_assoc_Δ_term`

La prueba sigue el patrón de inducción interna, pero adaptado para términos no necesariamente cerrados mediante la elección de una variable fresca v que no ocurre en a, b ni c.

1. Variable fresca: Se elige $v := \max_var_term(a) + \max_var_term(b) + \max_var_term(c) + 1$, garantizando que v no ocurre en ninguno de los tres términos.

2. Caso base ($v := 0$): Se demuestra $(a + b) + 0 = a + (b + 0)$ usando Ax0_{a+b} y Ax0_b junto con congruencia de la adición.

3. Paso inductivo ($v \to S(v)$): Se asume $(a + b) + v = a + (b + v)$ y se demuestra $(a + b) + S(v) = a + (b + S(v))$ usando $\text{Ax1}_{a+b}(v)$, $\text{Ax1}_b(v)$ y $\text{Ax1}_a(b + v)$ (versión modificada del axioma `add_succ` que suma sobre a con segundo argumento $b + v$).

4. Instanciación y descarga de axiomas: La inducción proporciona $\forall v\ \varphi'(v)$. Instanciando con c (verificando que c es libre para v en φ'), se obtiene $\varphi'(c) \in \Delta$. Las premisas se descargan usando `h_add_zero` y `h_add_succ`, junto con el argumento estándar de maximalidad y Henkin para las versiones cuantificadas universalmente. Tras eliminar todas las implicaciones, se obtiene $(a + b) + c = a + (b + c) \in \Delta$.

Comparación con `add_assoc_Δ`

La versión anterior `add_assoc_Δ` requería que los tres términos fueran cerrados (`TermClosed s`). Esta restricción se debía a que la demostración utilizaba

repetidamente `NotOccur_of_isClosed` para justificar las conmutaciones de sustituciones. La nueva versión `add_assoc_Δ_term` elimina esta restricción utilizando el lema `NotOccur_above_max_var_term` para garantizar la frescura de v respecto a términos arbitrarios. La construcción de los axiomas Ax1_a también se adapta: en lugar de usar `add_succ_axiom a (ArithTerm.var v)` (que daría $a + S(v) = S(a + v)$), se usa `add_succ_axiom a (ArithTerm.add b (ArithTerm.var v))`, que proporciona $a + S(b + v) = S(a + (b + v))$, que es la forma necesaria para el paso inductivo de la asociatividad.

Significado

`add_assoc_Δ_term` es la versión definitiva de la asociatividad en Δ, aplicable a cualquier terna de términos sin restricciones de clausura. Esta generalización es necesaria para las manipulaciones algebraicas en `NumForm` y otras fórmulas complejas, donde los términos involucrados pueden contener variables libres (como las variables ligadas de `NumForm`). La técnica de elegir una variable fresca que no ocurre en ninguno de los términos involucrados es un patrón general que permite extender propiedades demostradas para términos cerrados a términos arbitrarios, siempre que la propiedad sea "suficientemente uniforme" en la variable de inducción.

```
/--
   Lema auxiliar de sustitución uniforme para la asociatividad aditiva genérica.
   Demuestra que la sustitución de v por u sobre φ' solo reemplaza v en la relación
   atómica principal, preservando de forma exacta las premisas axiomáticas.
-/
lemma substForm_phi_prime_assoc {s : Nat} (a b : ArithTerm s) (v : Nat)
   (hva : NotOccur v a) (hvb : NotOccur v b) (u : ArithTerm s) :
   substForm v u
     (ArithForm.impl (add_zero_axiom (ArithTerm.add a b))
```

(ArithForm.impl (add_zero_axiom b)

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom (ArithTerm.add a b) (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom b (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom a (ArithTerm.add b (ArithTerm.var v))))

(ArithForm.eq (ArithTerm.add (ArithTerm.add a b) (ArithTerm.var v)) (ArithTerm.add a (ArithTerm.add b (ArithTerm.var v)))))))) =

ArithForm.impl (add_zero_axiom (ArithTerm.add a b))

(ArithForm.impl (add_zero_axiom b)

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom (ArithTerm.add a b) (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom b (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom a (ArithTerm.add b (ArithTerm.var v))))

(ArithForm.eq (ArithTerm.add (ArithTerm.add a b) u) (ArithTerm.add a (ArithTerm.add b u)))))) := by

dsimp [add_zero_axiom, add_succ_axiom, substForm, substTerm]

rw [substTerm_fresh a v u hva]

rw [substTerm_fresh b v u hvb]

have h_if : ∀ {α : Type} (x y : α), (if v = v then x else y) = x := fun x y => if_pos rfl

repeat rw [h_if]

-- Asociatividad de la Adición para Términos Objeto Genéricos en Δ: (a + b) + c = a + (b + c)

```
lemma add_assoc_Δ_term {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

  (a b c : ArithTerm s) :

  ArithForm.eq (ArithTerm.add (ArithTerm.add a b) c) (ArithTerm.add a (ArithTerm.add b c)) ∈ Δ := by

  let v := max_var_term a + max_var_term b + max_var_term c + 1

  have hva : NotOccur v a := NotOccur_above_max_var_term a v (by omega)

  have hvb : NotOccur v b := NotOccur_above_max_var_term b v (by omega)

  have hvc : NotOccur v c := NotOccur_above_max_var_term c v (by omega)


  let Ax0_ab : ArithForm s := add_zero_axiom (ArithTerm.add a b)

  let Ax0_b : ArithForm s := add_zero_axiom b

  let Ax1_ab_body : ArithForm s := add_succ_axiom (ArithTerm.add a b) (ArithTerm.var v)

  let Ax1_ab_forall : ArithForm s := ArithForm.forall_form v Ax1_ab_body

  let Ax1_b_body : ArithForm s := add_succ_axiom b (ArithTerm.var v)

  let Ax1_b_forall : ArithForm s := ArithForm.forall_form v Ax1_b_body

  let Ax1_a_body : ArithForm s := add_succ_axiom a (ArithTerm.add b (ArithTerm.var v))

  let Ax1_a_forall : ArithForm s := ArithForm.forall_form v Ax1_a_body

  let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.add (ArithTerm.add a b) (ArithTerm.var v)) (ArithTerm.add a (ArithTerm.add b (ArithTerm.var v)))


  let φ' : ArithForm s := ArithForm.impl Ax0_ab (ArithForm.impl Ax0_b (ArithForm.impl Ax1_ab_forall (ArithForm.impl Ax1_b_forall (ArithForm.impl Ax1_a_forall Eq_v))))


  have h_base : SPA_Deduction_Internal s ∅ (substForm v ArithTerm.zero φ') := by

    rw [substForm_phi_prime_assoc a b v hva hvb ArithTerm.zero]
```

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

let Γ5 : Set (ArithForm s) := insert Ax1_a_forall (insert Ax1_b_forall (insert Ax1_ab_forall (insert Ax0_b (insert Ax0_ab ∅))))

have h_Ax0_ab : SPA_Deduction_Internal s Γ5 Ax0_ab := SPA_Deduction_Internal.hyp Γ5 Ax0_ab (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))

have h_Ax0_b : SPA_Deduction_Internal s Γ5 Ax0_b := SPA_Deduction_Internal.hyp Γ5 Ax0_b (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

have h_refl_a : SPA_Deduction_Internal s Γ5 (ArithForm.eq a a) := SPA_Deduction_Internal.eq_refl Γ5 a

have h_congr : SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.add a (ArithTerm.add b ArithTerm.zero)) (ArithTerm.add a b)) :=

SPA_Deduction_Internal.eq_add Γ5 _ _ _ _ h_refl_a h_Ax0_b

have h_symm : SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.add a b) (ArithTerm.add a (ArithTerm.add b ArithTerm.zero))) :=

SPA_Deduction_Internal.eq_symm Γ5 _ _ h_congr

exact SPA_Deduction_Internal.eq_trans Γ5 _ _ _ h_Ax0_ab h_symm

have h_step : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v (ArithForm.impl φ' (substForm v (ArithTerm.succ (ArithTerm.var v)) φ'))) := by

apply SPA_Deduction_Internal.forall_intro

· rw [substForm_phi_prime_assoc a b v hva hvb (ArithTerm.succ (ArithTerm.var v))]

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

apply SPA_Deduction_Internal.imp_intro

let Γ6 : Set (ArithForm s) := insert Ax1_a_forall (insert Ax1_b_forall (insert Ax1_ab_forall (insert Ax0_b (insert Ax0_ab (insert φ' ∅)))))

have h_φ' : SPA_Deduction_Internal s Γ6 φ' := SPA_Deduction_Internal.hyp Γ6 φ' (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl))))))

have h_Ax0_ab : SPA_Deduction_Internal s Γ6 Ax0_ab := SPA_Deduction_Internal.hyp Γ6 Ax0_ab (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))

have h_Ax0_b : SPA_Deduction_Internal s Γ6 Ax0_b := SPA_Deduction_Internal.hyp Γ6 Ax0_b (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

have h_Ax1_ab : SPA_Deduction_Internal s Γ6 Ax1_ab_forall := SPA_Deduction_Internal.hyp Γ6 Ax1_ab_forall (Or.inr (Or.inr (Or.inl rfl)))

have h_Ax1_b : SPA_Deduction_Internal s Γ6 Ax1_b_forall := SPA_Deduction_Internal.hyp Γ6 Ax1_b_forall (Or.inr (Or.inl rfl))

have h_Ax1_a : SPA_Deduction_Internal s Γ6 Ax1_a_forall := SPA_Deduction_Internal.hyp Γ6 Ax1_a_forall (Or.inl rfl)

have h_imp1 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_φ' h_Ax0_ab

have h_imp2 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp1 h_Ax0_b

have h_imp3 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp2 h_Ax1_ab

have h_imp4 := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp3 h_Ax1_b

have h_Eq_v := SPA_Deduction_Internal.imp_elim Γ6 _ _ h_imp4 h_Ax1_a

have h_ax1_ab_v := SPA_Deduction_Internal.forall_elim Γ6 v Ax1_ab_body (ArithTerm.var v) h_Ax1_ab True.intro

change SPA_Deduction_Internal s Γ6 (ArithForm.eq (ArithTerm.add (ArithTerm.add (substTerm v (ArithTerm.var v) a) (substTerm v (ArithTerm.var v) b)) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add (ArithTerm.add (substTerm v (ArithTerm.var v) a) (substTerm v (ArithTerm.var v) b)) (substTerm v (ArithTerm.var v) (ArithTerm.var v))))) at h_ax1_ab_v

rw [substTerm_fresh a v (ArithTerm.var v) hva, substTerm_fresh b v (ArithTerm.var v) hvb, substTerm_var_same v (ArithTerm.var v)] at h_ax1_ab_v

have h_succ_hyp := SPA_Deduction_Internal.eq_succ Γ6 _ _ h_Eq_v

have h_ax1_b_v := SPA_Deduction_Internal.forall_elim Γ6 v Ax1_b_body (ArithTerm.var v) h_Ax1_b True.intro

change SPA_Deduction_Internal s Γ6 (ArithForm.eq (ArithTerm.add (substTerm v (ArithTerm.var v) b) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add (substTerm v (ArithTerm.var v) b) (substTerm v (ArithTerm.var v) (ArithTerm.var v))))) at h_ax1_b_v

rw [substTerm_fresh b v (ArithTerm.var v) hvb, substTerm_var_same v (ArithTerm.var v)] at h_ax1_b_v

have h_refl_a : SPA_Deduction_Internal s Γ6 (ArithForm.eq a a) := SPA_Deduction_Internal.eq_refl Γ6 a

have h_congr1 := SPA_Deduction_Internal.eq_add Γ6 _ _ _ _ h_refl_a h_ax1_b_v

have h_ax1_a_v := SPA_Deduction_Internal.forall_elim Γ6 v Ax1_a_body (ArithTerm.var v) h_Ax1_a True.intro

have h_subst_b_v : substTerm v (ArithTerm.var v) (ArithTerm.add b (ArithTerm.var v)) = ArithTerm.add b (ArithTerm.var v) := by

dsimp [substTerm]

rw [substTerm_fresh b v (ArithTerm.var v) hvb]

have h_if : (if v = v then (ArithTerm.var v : ArithTerm s) else (ArithTerm.var v : ArithTerm s)) = (ArithTerm.var v : ArithTerm s) := if_pos rfl

rw [h_if]

change SPA_Deduction_Internal s Γ6 (ArithForm.eq (ArithTerm.add (substTerm v (ArithTerm.var v) a) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.add b (ArithTerm.var v))))) (ArithTerm.succ (ArithTerm.add (substTerm v (ArithTerm.var v) a) (substTerm v (ArithTerm.var v) (ArithTerm.add b (ArithTerm.var v)))))) at h_ax1_a_v

rw [substTerm_fresh a v (ArithTerm.var v) hva, h_subst_b_v] at h_ax1_a_v

have h_lhs := SPA_Deduction_Internal.eq_trans Γ6 _ _ _ h_ax1_ab_v h_succ_hyp

have h_rhs := SPA_Deduction_Internal.eq_trans Γ6 _ _ _ h_congr1 h_ax1_a_v

have h_rhs_symm := SPA_Deduction_Internal.eq_symm Γ6 _ _ h_rhs

exact SPA_Deduction_Internal.eq_trans Γ6 _ _ _ h_lhs h_rhs_symm

```
  · intro ψ hψ
    nomatch hψ

  have h_ind : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ') :=
    SPA_Deduction_Internal.induction ∅ φ' v h_base h_step

  have h_ind_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form v φ') :=
    SPA_Deduction_Internal.weaken ∅ _ Δ h_ind (Set.empty_subset _)

  have h_free_phi' : IsFreeFor v c φ' := by
    dsimp [φ', Ax0_ab, Ax0_b, Ax1_ab_forall, Ax1_b_forall, Ax1_a_forall, Eq_v,
add_zero_axiom, add_succ_axiom, IsFreeFor]
    have h1 : (if v = v then True else NotOccur v c ∧ IsFreeFor v c Ax1_ab_body) = True
:= if_pos rfl
    have h2 : (if v = v then True else NotOccur v c ∧ IsFreeFor v c Ax1_b_body) = True :=
if_pos rfl
    have h3 : (if v = v then True else NotOccur v c ∧ IsFreeFor v c Ax1_a_body) = True :=
if_pos rfl
    rw [h1, h2, h3]
    exact ⟨True.intro, True.intro, True.intro, True.intro, True.intro, True.intro⟩

  have h_elim := SPA_Deduction_Internal.forall_elim Δ v φ' c h_ind_Δ h_free_phi'
  rw [substForm_phi_prime_assoc a b v hva hvb c] at h_elim

  have h_Ax0_ab_Δ : SPA_Deduction_Internal s Δ Ax0_ab := SPA_Deduction_Internal.hyp
Δ _ (h_add_zero (ArithTerm.add a b))
  have h_Ax0_b_Δ : SPA_Deduction_Internal s Δ Ax0_b := SPA_Deduction_Internal.hyp Δ
_ (h_add_zero b)
  have h_elim1 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim h_Ax0_ab_Δ
  have h_elim2 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim1 h_Ax0_b_Δ
```

```
  have h_Ax1_ab_in : Ax1_ab_forall ∈ Δ := by

    by_contra h_not

    have h_neg : ArithForm.neg Ax1_ab_forall ∈ Δ := by cases h_max Ax1_ab_forall with | inl h' => contradiction | inr h' => exact h'

    have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_ab_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]

    obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_ab_body) h_nd

    have h_subst_ax : substForm v t.val Ax1_ab_body =

      ArithForm.eq (ArithTerm.add (ArithTerm.add a b) (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add (ArithTerm.add a b) t.val)) := by

      dsimp [Ax1_ab_body, add_succ_axiom, substForm, substTerm]

      rw [substTerm_fresh a v t.val hva, substTerm_fresh b v t.val hvb]

      have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl

      rw [h_if]

    have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.add (ArithTerm.add a b) (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add (ArithTerm.add a b) t.val))) ∈ Δ := by

      change ArithForm.neg (substForm v t.val Ax1_ab_body) ∈ Δ at ht

      rw [h_subst_ax] at ht

      exact ht

    have h_pos := h_add_succ (ArithTerm.add a b) t.val

    exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))


  have h_Ax1_b_in : Ax1_b_forall ∈ Δ := by

    by_contra h_not

    have h_neg : ArithForm.neg Ax1_b_forall ∈ Δ := by cases h_max Ax1_b_forall with | inl h' => contradiction | inr h' => exact h'

    have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_b_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
```

```
  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_b_body) h_nd
  have h_subst_ax : substForm v t.val Ax1_b_body =
    ArithForm.eq (ArithTerm.add b (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add b t.val)) := by
    dsimp [Ax1_b_body, add_succ_axiom, substForm, substTerm]
    rw [substTerm_fresh b v t.val hvb]
    have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
    rw [h_if]
  have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.add b (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add b t.val))) ∈ Δ := by
    change ArithForm.neg (substForm v t.val Ax1_b_body) ∈ Δ at ht
    rw [h_subst_ax] at ht
    exact ht
  have h_pos := h_add_succ b t.val
  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))


 have h_Ax1_a_in : Ax1_a_forall ∈ Δ := by
  by_contra h_not
  have h_neg : ArithForm.neg Ax1_a_forall ∈ Δ := by cases h_max Ax1_a_forall with | inl h' => contradiction | inr h' => exact h'
  have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_a_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_a_body) h_nd
  have h_subst_ax : substForm v t.val Ax1_a_body =
    ArithForm.eq (ArithTerm.add a (ArithTerm.succ (ArithTerm.add b t.val))) (ArithTerm.succ (ArithTerm.add a (ArithTerm.add b t.val))) := by
    dsimp [Ax1_a_body, add_succ_axiom, substForm, substTerm]
    rw [substTerm_fresh a v t.val hva, substTerm_fresh b v t.val hvb]
    have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
```

```
    rw [h_if]
  have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.add a (ArithTerm.succ (ArithTerm.add b t.val))) (ArithTerm.succ (ArithTerm.add a (ArithTerm.add b t.val)))) ∈ Δ := by
    change ArithForm.neg (substForm v t.val Ax1_a_body) ∈ Δ at ht
    rw [h_subst_ax] at ht
    exact ht
  have h_pos := h_add_succ a (ArithTerm.add b t.val)
  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))

 have h_Ax1_ab_Δ : SPA_Deduction_Internal s Δ Ax1_ab_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_ab_in
 have h_Ax1_b_Δ : SPA_Deduction_Internal s Δ Ax1_b_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_b_in
 have h_Ax1_a_Δ : SPA_Deduction_Internal s Δ Ax1_a_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_a_in

 have h_elim3 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim2 h_Ax1_ab_Δ
 have h_elim4 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim3 h_Ax1_b_Δ
 have h_final_ded := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim4 h_Ax1_a_Δ

 exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final_ded
```

Lema `succ_mul_Δ`: Distributividad izquierda del sucesor en la multiplicación

Se demuestra una propiedad algebraica fundamental que conecta el sucesor con la multiplicación dentro de Δ: para cualesquiera términos x e y, la

multiplicación del sucesor de x por y es demostrablemente igual a $x \cdot y$ más y:

$$ \forall x, y \in \text{ArithTerm } s,\ \Delta \vdash S(x) \cdot y = x \cdot y + y $$

Lema auxiliar `substForm_phi_prime_succ_mul`

Define la fórmula de inducción para esta propiedad y demuestra su comportamiento bajo sustitución. La fórmula $\varphi'(v)$ es notablemente más compleja que las anteriores, conteniendo nueve premisas encadenadas que incluyen:

- Axiomas de cero para la multiplicación: `mul_zero` para $S(x)$ y para x
- Axioma de cero para la adición: `add_zero` para $x \cdot 0$
- Axiomas de sucesor para la multiplicación: `mul_succ` para $S(x)$ y para x
- Axiomas de sucesor para la adición: `add_succ` para $(S(x) \cdot v) + x$ y para $(x \cdot v + x) + v$
- Conmutatividad/asociatividad: $(x \cdot v + v) + x = (x \cdot v + x) + v$

La conclusión es $S(x) \cdot v = x \cdot v + v$.

Estructura de la demostración de `succ_mul_Δ`

La prueba sigue el patrón de inducción interna pero con una complejidad significativamente mayor debido a la cantidad de propiedades algebraicas involucradas.

1. Variable fresca: Se elige v que no ocurre en x ni en y.

2. Caso base ($v := 0$): Se demuestra $S(x) \cdot 0 = x \cdot 0 + 0$ usando los axiomas de cero para la multiplicación y la adición. Ambos lados se reducen a 0 mediante `mul_zero_axiom` y `add_zero_axiom`.

3. Paso inductivo ($v \to S(v)$): La cadena de igualdades es particularmente elaborada:

- Por `mul_succ` para $S(x)$: $S(x) \cdot S(v) = S(x) \cdot v + S(x)$

- Por `add_succ` para $(S(x) \cdot v) + x$: $S(x) \cdot v + S(x) = S(S(x) \cdot v + x)$

- Por hipótesis de inducción $S(x) \cdot v = x \cdot v + v$ y sumando x: $S(S(x) \cdot v + x) = S((x \cdot v + v) + x)$

- Por conmutatividad/asociatividad: $S((x \cdot v + v) + x) = S((x \cdot v + x) + v)$

- Por `add_succ` para $(x \cdot v + x) + v$: $S((x \cdot v + x) + v) = (x \cdot v + x) + S(v)$

- Por `mul_succ` para x (invertido): $(x \cdot v + x) + S(v) = x \cdot S(v) + S(v)$

Cada paso se justifica mediante una regla del sistema deductivo o un lema algebraico previamente demostrado.

4. Verificación de pertenencia de axiomas cuantificados: Cinco fórmulas cuantificadas universalmente requieren el argumento estándar de maximalidad y Henkin. Para la última (`Eq_swap_forall`), que expresa $(x \cdot v + v) + x = (x \cdot v + x) + v$, la verificación utiliza `add_assoc_Δ_term` y `add_comm_Δ` para demostrar que efectivamente la igualdad es derivable en Δ, por lo que su negación no puede pertenecer a Δ.

Significado

`succ_mul_Δ` es una de las propiedades algebraicas más profundas demostradas hasta ahora en Δ. Establece la distributividad del sucesor sobre la multiplicación, que es un ingrediente esencial para demostrar propiedades más avanzadas como la conmutatividad de la multiplicación o la representabilidad de funciones recursivas.

La complejidad de la demostración refleja un fenómeno bien conocido en la aritmética formal: propiedades que son inmediatas en el modelo estándar (como $S(x) \cdot y = x \cdot y + y$) requieren demostraciones extensas cuando se trabaja puramente con los axiomas de Peano y las reglas de inferencia. La necesidad de incluir la conmutatividad/asociatividad como una premisa adicional (`Eq_swap_forall`) y verificarla independientemente muestra cómo las demostraciones internas a menudo requieren "empaquetar" lemas auxiliares dentro de la fórmula de inducción.

Este lema será utilizado en la demostración de la funcionalidad de `NumForm`, donde aparecerán expresiones de la forma $S(\overline{k}) \cdot \overline{M}$ que deberán ser manipuladas algebraicamente. La capacidad de distribuir el sucesor sobre la multiplicación es crucial para simplificar tales expresiones y reducirlas a formas donde los lemas de representabilidad previos puedan aplicarse.

```
/--
  Lema auxiliar de sustitución uniforme para la distributividad del sucesor en la multiplicación.
  Demuestra de forma ultra-robusta que la sustitución de v por u sobre φ'
  solo reemplaza v en la relación atómica principal, preservando las premisas axiomáticas.
-/
lemma substForm_phi_prime_succ_mul {s : Nat} (x : ArithTerm s) (v : Nat)
  (hvx : NotOccur v x) (u : ArithTerm s) :
  substForm v u
    (ArithForm.impl (mul_zero_axiom (ArithTerm.succ x))
      (ArithForm.impl (mul_zero_axiom x)
        (ArithForm.impl (add_zero_axiom (ArithTerm.mul x ArithTerm.zero))
```

(ArithForm.impl (ArithForm.forall_form v (mul_succ_axiom (ArithTerm.succ x) (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (mul_succ_axiom x (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom (ArithTerm.mul (ArithTerm.succ x) (ArithTerm.var v)) x))

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) x) (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) (ArithTerm.var v)) x) (ArithTerm.add (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) x) (ArithTerm.var v))))

(ArithForm.eq (ArithTerm.mul (ArithTerm.succ x) (ArithTerm.var v)) (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) (ArithTerm.var v))))))))))) =

ArithForm.impl (mul_zero_axiom (ArithTerm.succ x))

(ArithForm.impl (mul_zero_axiom x)

(ArithForm.impl (add_zero_axiom (ArithTerm.mul x ArithTerm.zero))

(ArithForm.impl (ArithForm.forall_form v (mul_succ_axiom (ArithTerm.succ x) (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (mul_succ_axiom x (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom (ArithTerm.mul (ArithTerm.succ x) (ArithTerm.var v)) x))

(ArithForm.impl (ArithForm.forall_form v (add_succ_axiom (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) x) (ArithTerm.var v)))

(ArithForm.impl (ArithForm.forall_form v (ArithForm.eq (ArithTerm.add (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) (ArithTerm.var v)) x) (ArithTerm.add (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) x) (ArithTerm.var v))))

(ArithForm.eq (ArithTerm.mul (ArithTerm.succ x) u) (ArithTerm.add (ArithTerm.mul x u) u)))))))))) := by

dsimp [mul_zero_axiom, add_zero_axiom, mul_succ_axiom, add_succ_axiom, substForm, substTerm]

rw [substTerm_fresh x v u hvx]

have h_if : ∀ {α : Type} (x y : α), (if v = v then x else y) = x := fun x y => if_pos rfl

```
  repeat rw [h_if]



/--

   Propiedad Distributiva por la Izquierda del Sucesor en la Multiplicación en Δ.

   S(x) · y = x · y + y

-/

lemma succ_mul_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

   (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

   (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

   (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

   (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

   (x y : ArithTerm s) :

   ArithForm.eq (ArithTerm.mul (ArithTerm.succ x) y) (ArithTerm.add (ArithTerm.mul x y) y) ∈ Δ := by

  let v := max_var_term x + max_var_term y + 1

  have hvx : NotOccur v x := NotOccur_above_max_var_term x v (by omega)

  have hvy : NotOccur v y := NotOccur_above_max_var_term y v (by omega)


  let Ax0_sx : ArithForm s := mul_zero_axiom (ArithTerm.succ x)

  let Ax0_x : ArithForm s := mul_zero_axiom x

  let Ax0_a0 : ArithForm s := add_zero_axiom (ArithTerm.mul x ArithTerm.zero)

  let Ax1_sx_body : ArithForm s := mul_succ_axiom (ArithTerm.succ x) (ArithTerm.var v)

  let Ax1_sx_forall : ArithForm s := ArithForm.forall_form v Ax1_sx_body

  let Ax1_x_body : ArithForm s := mul_succ_axiom x (ArithTerm.var v)

  let Ax1_x_forall : ArithForm s := ArithForm.forall_form v Ax1_x_body

  let Ax_A1_1_body : ArithForm s := add_succ_axiom (ArithTerm.mul (ArithTerm.succ x) (ArithTerm.var v)) x
```

```
  let Ax_A1_1_forall : ArithForm s := ArithForm.forall_form v Ax_A1_1_body

  let Ax_A1_2_body : ArithForm s := add_succ_axiom (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) x) (ArithTerm.var v)

  let Ax_A1_2_forall : ArithForm s := ArithForm.forall_form v Ax_A1_2_body

  let Eq_swap_body : ArithForm s := ArithForm.eq (ArithTerm.add (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) (ArithTerm.var v)) x) (ArithTerm.add (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) x) (ArithTerm.var v))

  let Eq_swap_forall : ArithForm s := ArithForm.forall_form v Eq_swap_body

  let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.mul (ArithTerm.succ x) (ArithTerm.var v)) (ArithTerm.add (ArithTerm.mul x (ArithTerm.var v)) (ArithTerm.var v))


  let φ' : ArithForm s := ArithForm.impl Ax0_sx (ArithForm.impl Ax0_x (ArithForm.impl Ax0_a0 (ArithForm.impl Ax1_sx_forall (ArithForm.impl Ax1_x_forall (ArithForm.impl Ax_A1_1_forall (ArithForm.impl Ax_A1_2_forall (ArithForm.impl Eq_swap_forall Eq_v)))))))


  have h_base : SPA_Deduction_Internal s ∅ (substForm v ArithTerm.zero φ') := by

    rw [substForm_phi_prime_succ_mul x v hvx ArithTerm.zero]

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    apply SPA_Deduction_Internal.imp_intro

    let Γ8 : Set (ArithForm s) := insert Eq_swap_forall (insert Ax_A1_2_forall (insert Ax_A1_1_forall (insert Ax1_x_forall (insert Ax1_sx_forall (insert Ax0_a0 (insert Ax0_x (insert Ax0_sx ∅))))))))

    have h_Ax0_sx : SPA_Deduction_Internal s Γ8 Ax0_sx := SPA_Deduction_Internal.hyp Γ8 Ax0_sx (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))))))
```

```
  have h_Ax0_x : SPA_Deduction_Internal s Γ8 Ax0_x := SPA_Deduction_Internal.hyp Γ8 Ax0_x (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))))

  have h_Ax0_a0 : SPA_Deduction_Internal s Γ8 Ax0_a0 := SPA_Deduction_Internal.hyp Γ8 Ax0_a0 (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl))))))


  have h_symm_x := SPA_Deduction_Internal.eq_symm Γ8 _ _ h_Ax0_x

  have h_symm_a0 := SPA_Deduction_Internal.eq_symm Γ8 _ _ h_Ax0_a0


  have h_t1 := SPA_Deduction_Internal.eq_trans Γ8 _ _ _ h_Ax0_sx h_symm_x

  exact SPA_Deduction_Internal.eq_trans Γ8 _ _ _ h_t1 h_symm_a0


 have h_step : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v (ArithForm.impl φ' (substForm v (ArithTerm.succ (ArithTerm.var v)) φ'))) := by

  apply SPA_Deduction_Internal.forall_intro

  · rw [substForm_phi_prime_succ_mul x v hvx (ArithTerm.succ (ArithTerm.var v))]

   apply SPA_Deduction_Internal.imp_intro

   apply SPA_Deduction_Internal.imp_intro

   apply SPA_Deduction_Internal.imp_intro

   apply SPA_Deduction_Internal.imp_intro

   apply SPA_Deduction_Internal.imp_intro

   apply SPA_Deduction_Internal.imp_intro

   apply SPA_Deduction_Internal.imp_intro

   apply SPA_Deduction_Internal.imp_intro

   apply SPA_Deduction_Internal.imp_intro

   let Γ9 : Set (ArithForm s) := insert Eq_swap_forall (insert Ax_A1_2_forall (insert Ax_A1_1_forall (insert Ax1_x_forall (insert Ax1_sx_forall (insert Ax0_a0 (insert Ax0_x (insert Ax0_sx (insert φ' ∅)))))))

   have h_φ' : SPA_Deduction_Internal s Γ9 φ' := SPA_Deduction_Internal.hyp Γ9 φ' (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl))))))))
```

have h_Ax0_sx : SPA_Deduction_Internal s Γ9 Ax0_sx := SPA_Deduction_Internal.hyp Γ9 Ax0_sx (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))))

have h_Ax0_x : SPA_Deduction_Internal s Γ9 Ax0_x := SPA_Deduction_Internal.hyp Γ9 Ax0_x (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))))

have h_Ax0_a0 : SPA_Deduction_Internal s Γ9 Ax0_a0 := SPA_Deduction_Internal.hyp Γ9 Ax0_a0 (Or.inr (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl))))))

have h_Ax1_sx : SPA_Deduction_Internal s Γ9 Ax1_sx_forall := SPA_Deduction_Internal.hyp Γ9 Ax1_sx_forall (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))

have h_Ax1_x : SPA_Deduction_Internal s Γ9 Ax1_x_forall := SPA_Deduction_Internal.hyp Γ9 Ax1_x_forall (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

have h_Ax_A1_1 : SPA_Deduction_Internal s Γ9 Ax_A1_1_forall := SPA_Deduction_Internal.hyp Γ9 Ax_A1_1_forall (Or.inr (Or.inr (Or.inl rfl)))

have h_Ax_A1_2 : SPA_Deduction_Internal s Γ9 Ax_A1_2_forall := SPA_Deduction_Internal.hyp Γ9 Ax_A1_2_forall (Or.inr (Or.inl rfl))

have h_Eq_swap : SPA_Deduction_Internal s Γ9 Eq_swap_forall := SPA_Deduction_Internal.hyp Γ9 Eq_swap_forall (Or.inl rfl)

have h_imp1 := SPA_Deduction_Internal.imp_elim Γ9 _ _ h_φ' h_Ax0_sx

have h_imp2 := SPA_Deduction_Internal.imp_elim Γ9 _ _ h_imp1 h_Ax0_x

have h_imp3 := SPA_Deduction_Internal.imp_elim Γ9 _ _ h_imp2 h_Ax0_a0

have h_imp4 := SPA_Deduction_Internal.imp_elim Γ9 _ _ h_imp3 h_Ax1_sx

have h_imp5 := SPA_Deduction_Internal.imp_elim Γ9 _ _ h_imp4 h_Ax1_x

have h_imp6 := SPA_Deduction_Internal.imp_elim Γ9 _ _ h_imp5 h_Ax_A1_1

have h_imp7 := SPA_Deduction_Internal.imp_elim Γ9 _ _ h_imp6 h_Ax_A1_2

have h_Hind := SPA_Deduction_Internal.imp_elim Γ9 _ _ h_imp7 h_Eq_swap

have h_Ax1_sx_v := SPA_Deduction_Internal.forall_elim Γ9 v Ax1_sx_body (ArithTerm.var v) h_Ax1_sx True.intro

change SPA_Deduction_Internal s Γ9 (ArithForm.eq (ArithTerm.mul (ArithTerm.succ (substTerm v (ArithTerm.var v) x)) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.add (ArithTerm.mul

(ArithTerm.succ (substTerm v (ArithTerm.var v) x)) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (ArithTerm.succ (substTerm v (ArithTerm.var v) x)))) at h_Ax1_sx_v

rw [substTerm_fresh x v (ArithTerm.var v) hvx, substTerm_var_same v (ArithTerm.var v)] at h_Ax1_sx_v

have h_Ax_A1_1_v := SPA_Deduction_Internal.forall_elim Γ9 v Ax_A1_1_body (ArithTerm.var v) h_Ax_A1_1 True.intro

change SPA_Deduction_Internal s Γ9 (ArithForm.eq (ArithTerm.add (ArithTerm.mul (ArithTerm.succ (substTerm v (ArithTerm.var v) x)) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (ArithTerm.succ (substTerm v (ArithTerm.var v) x))) (ArithTerm.succ (ArithTerm.add (ArithTerm.mul (ArithTerm.succ (substTerm v (ArithTerm.var v) x)) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) x)))) at h_Ax_A1_1_v

rw [substTerm_fresh x v (ArithTerm.var v) hvx, substTerm_var_same v (ArithTerm.var v)] at h_Ax_A1_1_v

have h_refl_x : SPA_Deduction_Internal s Γ9 (ArithForm.eq x x) := SPA_Deduction_Internal.eq_refl Γ9 x

have h_congr_Hind := SPA_Deduction_Internal.eq_add Γ9 _ _ _ _ h_Hind h_refl_x

have h_succ_Hind := SPA_Deduction_Internal.eq_succ Γ9 _ _ h_congr_Hind

have h_Eq_swap_v := SPA_Deduction_Internal.forall_elim Γ9 v Eq_swap_body (ArithTerm.var v) h_Eq_swap True.intro

change SPA_Deduction_Internal s Γ9 (ArithForm.eq (ArithTerm.add (ArithTerm.add (ArithTerm.mul (substTerm v (ArithTerm.var v) x) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) x)) (ArithTerm.add (ArithTerm.add (ArithTerm.mul (substTerm v (ArithTerm.var v) x) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) x)) (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) at h_Eq_swap_v

rw [substTerm_fresh x v (ArithTerm.var v) hvx, substTerm_var_same v (ArithTerm.var v)] at h_Eq_swap_v

have h_succ_Eq_swap := SPA_Deduction_Internal.eq_succ Γ9 _ _ h_Eq_swap_v

have h_Ax_A1_2_v := SPA_Deduction_Internal.forall_elim Γ9 v Ax_A1_2_body (ArithTerm.var v) h_Ax_A1_2 True.intro

change SPA_Deduction_Internal s Γ9 (ArithForm.eq (ArithTerm.add (ArithTerm.add (ArithTerm.mul (substTerm v (ArithTerm.var v) x) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) x)) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.succ (ArithTerm.add (ArithTerm.add (ArithTerm.mul (substTerm v (ArithTerm.var v) x) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) x)) (substTerm v (ArithTerm.var v) (ArithTerm.var v))))) at h_Ax_A1_2_v

rw [substTerm_fresh x v (ArithTerm.var v) hvx, substTerm_var_same v (ArithTerm.var v)] at h_Ax_A1_2_v

have h_symm_A1_2 := SPA_Deduction_Internal.eq_symm Γ9 _ _ h_Ax_A1_2_v

have h_Ax1_x_v := SPA_Deduction_Internal.forall_elim Γ9 v Ax1_x_body (ArithTerm.var v) h_Ax1_x True.intro

change SPA_Deduction_Internal s Γ9 (ArithForm.eq (ArithTerm.mul (substTerm v (ArithTerm.var v) x) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.add (ArithTerm.mul (substTerm v (ArithTerm.var v) x) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) x))) at h_Ax1_x_v

rw [substTerm_fresh x v (ArithTerm.var v) hvx, substTerm_var_same v (ArithTerm.var v)] at h_Ax1_x_v

have h_symm_Ax1_x := SPA_Deduction_Internal.eq_symm Γ9 _ _ h_Ax1_x_v

have h_refl_sv : SPA_Deduction_Internal s Γ9 (ArithForm.eq (ArithTerm.succ (ArithTerm.var v)) (ArithTerm.succ (ArithTerm.var v))) := SPA_Deduction_Internal.eq_refl Γ9 (ArithTerm.succ (ArithTerm.var v))

have h_congr_Ax1_x := SPA_Deduction_Internal.eq_add Γ9 _ _ _ _ h_symm_Ax1_x h_refl_sv

have h_t1 := SPA_Deduction_Internal.eq_trans Γ9 _ _ _ h_Ax1_sx_v h_Ax_A1_1_v

have h_t2 := SPA_Deduction_Internal.eq_trans Γ9 _ _ _ h_t1 h_succ_Hind

have h_t3 := SPA_Deduction_Internal.eq_trans Γ9 _ _ _ h_t2 h_succ_Eq_swap

have h_t4 := SPA_Deduction_Internal.eq_trans Γ9 _ _ _ h_t3 h_symm_A1_2

have h_t5 := SPA_Deduction_Internal.eq_trans Γ9 _ _ _ h_t4 h_congr_Ax1_x

```
    exact h_t5
  · intro ψ hψ
    nomatch hψ

  have h_ind : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ') :=
    SPA_Deduction_Internal.induction ∅ φ' v h_base h_step

  have h_ind_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form v φ') :=
    SPA_Deduction_Internal.weaken ∅ _ Δ h_ind (Set.empty_subset _)

  have h_free_phi' : IsFreeFor v y φ' := by
    dsimp [φ', Ax0_sx, Ax0_x, Ax0_a0, Ax1_sx_forall, Ax1_x_forall, Ax_A1_1_forall,
Ax_A1_2_forall, Eq_swap_forall, Eq_v, mul_zero_axiom, add_zero_axiom,
mul_succ_axiom, add_succ_axiom, IsFreeFor]
    have h1 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Ax1_sx_body) = True
:= if_pos rfl
    have h2 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Ax1_x_body) = True :=
if_pos rfl
    have h3 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Ax_A1_1_body) = True
:= if_pos rfl
    have h4 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Ax_A1_2_body) = True
:= if_pos rfl
    have h5 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Eq_swap_body) = True
:= if_pos rfl
    rw [h1, h2, h3, h4, h5]
    repeat constructor

  have h_elim := SPA_Deduction_Internal.forall_elim Δ v φ' y h_ind_Δ h_free_phi'
  rw [substForm_phi_prime_succ_mul x v hvx y] at h_elim
```

```
  have h_Ax0_sx_Δ : SPA_Deduction_Internal s Δ Ax0_sx := SPA_Deduction_Internal.hyp Δ _ (h_mul_zero (ArithTerm.succ x))

  have h_Ax0_x_Δ : SPA_Deduction_Internal s Δ Ax0_x := SPA_Deduction_Internal.hyp Δ _ (h_mul_zero x)

  have h_Ax0_a0_Δ : SPA_Deduction_Internal s Δ Ax0_a0 := SPA_Deduction_Internal.hyp Δ _ (h_add_zero (ArithTerm.mul x ArithTerm.zero))


  have h_elim1 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim h_Ax0_sx_Δ

  have h_elim2 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim1 h_Ax0_x_Δ

  have h_elim3 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim2 h_Ax0_a0_Δ


  have h_Ax1_sx_in : Ax1_sx_forall ∈ Δ := by

    by_contra h_not

    have h_neg : ArithForm.neg Ax1_sx_forall ∈ Δ := by cases h_max Ax1_sx_forall with | inl h' => contradiction | inr h' => exact h'

    have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_sx_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]

    obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_sx_body) h_nd

    have h_subst_ax : substForm v t.val Ax1_sx_body =

      ArithForm.eq (ArithTerm.mul (ArithTerm.succ x) (ArithTerm.succ t.val)) (ArithTerm.add (ArithTerm.mul (ArithTerm.succ x) t.val) (ArithTerm.succ x)) := by

      dsimp [Ax1_sx_body, mul_succ_axiom, substForm, substTerm]

      rw [substTerm_fresh x v t.val hvx]

      have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl

      rw [h_if]

    have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.mul (ArithTerm.succ x) (ArithTerm.succ t.val)) (ArithTerm.add (ArithTerm.mul (ArithTerm.succ x) t.val) (ArithTerm.succ x))) ∈ Δ := by

      change ArithForm.neg (substForm v t.val Ax1_sx_body) ∈ Δ at ht

      rw [h_subst_ax] at ht

      exact ht
```

```
  have h_pos := h_mul_succ (ArithTerm.succ x) t.val
  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))

 have h_Ax1_x_in : Ax1_x_forall ∈ Δ := by
  by_contra h_not
  have h_neg : ArithForm.neg Ax1_x_forall ∈ Δ := by cases h_max Ax1_x_forall with | inl h' => contradiction | inr h' => exact h'
  have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_x_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_x_body) h_nd
  have h_subst_ax : substForm v t.val Ax1_x_body =
    ArithForm.eq (ArithTerm.mul x (ArithTerm.succ t.val)) (ArithTerm.add (ArithTerm.mul x t.val) x) := by
    dsimp [Ax1_x_body, mul_succ_axiom, substForm, substTerm]
    rw [substTerm_fresh x v t.val hvx]
    have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
    rw [h_if]
  have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.mul x (ArithTerm.succ t.val)) (ArithTerm.add (ArithTerm.mul x t.val) x)) ∈ Δ := by
    change ArithForm.neg (substForm v t.val Ax1_x_body) ∈ Δ at ht
    rw [h_subst_ax] at ht
    exact ht
  have h_pos := h_mul_succ x t.val
  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))

 have h_Ax_A1_1_in : Ax_A1_1_forall ∈ Δ := by
  by_contra h_not
  have h_neg : ArithForm.neg Ax_A1_1_forall ∈ Δ := by cases h_max Ax_A1_1_forall with | inl h' => contradiction | inr h' => exact h'
```

```
  have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax_A1_1_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]

  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax_A1_1_body) h_nd

  have h_subst_ax : substForm v t.val Ax_A1_1_body =

    ArithForm.eq (ArithTerm.add (ArithTerm.mul (ArithTerm.succ x) t.val) (ArithTerm.succ x)) (ArithTerm.succ (ArithTerm.add (ArithTerm.mul (ArithTerm.succ x) t.val) x)) := by

    dsimp [Ax_A1_1_body, add_succ_axiom, substForm, substTerm]

    rw [substTerm_fresh x v t.val hvx]

    have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl

    rw [h_if]

  have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.add (ArithTerm.mul (ArithTerm.succ x) t.val) (ArithTerm.succ x)) (ArithTerm.succ (ArithTerm.add (ArithTerm.mul (ArithTerm.succ x) t.val) x))) ∈ Δ := by

    change ArithForm.neg (substForm v t.val Ax_A1_1_body) ∈ Δ at ht

    rw [h_subst_ax] at ht

    exact ht

  have h_pos := h_add_succ (ArithTerm.mul (ArithTerm.succ x) t.val) x

  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))


 have h_Ax_A1_2_in : Ax_A1_2_forall ∈ Δ := by

  by_contra h_not

  have h_neg : ArithForm.neg Ax_A1_2_forall ∈ Δ := by cases h_max Ax_A1_2_forall with | inl h' => contradiction | inr h' => exact h'

  have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax_A1_2_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]

  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax_A1_2_body) h_nd

  have h_subst_ax : substForm v t.val Ax_A1_2_body =

    ArithForm.eq (ArithTerm.add (ArithTerm.add (ArithTerm.mul x t.val) x) (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add (ArithTerm.add (ArithTerm.mul x t.val) x) t.val)) := by
```

```
    dsimp [Ax_A1_2_body, add_succ_axiom, substForm, substTerm]
    rw [substTerm_fresh x v t.val hvx]
    have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
    rw [h_if]
  have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.add (ArithTerm.add (ArithTerm.mul x t.val) x) (ArithTerm.succ t.val)) (ArithTerm.succ (ArithTerm.add (ArithTerm.add (ArithTerm.mul x t.val) x) t.val))) ∈ Δ := by
    change ArithForm.neg (substForm v t.val Ax_A1_2_body) ∈ Δ at ht
    rw [h_subst_ax] at ht
    exact ht
  have h_pos := h_add_succ (ArithTerm.add (ArithTerm.mul x t.val) x) t.val
  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))


 have h_Eq_swap_in : Eq_swap_forall ∈ Δ := by
  by_contra h_not
  have h_neg : ArithForm.neg Eq_swap_forall ∈ Δ := by cases h_max Eq_swap_forall with | inl h' => contradiction | inr h' => exact h'
  have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Eq_swap_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Eq_swap_body) h_nd
  have h_subst_ax : substForm v t.val Eq_swap_body =
    ArithForm.eq (ArithTerm.add (ArithTerm.add (ArithTerm.mul x t.val) t.val) x) (ArithTerm.add (ArithTerm.add (ArithTerm.mul x t.val) x) t.val) := by
    dsimp [Eq_swap_body, substForm, substTerm]
    rw [substTerm_fresh x v t.val hvx]
    have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
    rw [h_if]
  have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.add (ArithTerm.add (ArithTerm.mul x t.val) t.val) x) (ArithTerm.add (ArithTerm.add (ArithTerm.mul x t.val) x) t.val)) ∈ Δ := by
```

```
    change ArithForm.neg (substForm v t.val Eq_swap_body) ∈ Δ at ht
    rw [h_subst_ax] at ht
    exact ht

  let A' := ArithTerm.mul x t.val
  let V' := t.val
  let X' := x
  have h_assoc1 := add_assoc_Δ_term Δ h_max h_con h_henkin h_add_zero h_add_succ A' V' X'
  have h_comm_VX := add_comm_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ V' X'
  have h_refl_A' := eq_refl_in_Δ Δ h_max h_con A'
  have h_congr_A' := eq_add_Δ h_max h_con h_refl_A' h_comm_VX
  have h_assoc2 := add_assoc_Δ_term Δ h_max h_con h_henkin h_add_zero h_add_succ A' X' V'
  have h_assoc2_symm := eq_symm_Δ h_max h_con h_assoc2

  have h_t1 := eq_trans_Δ h_max h_con h_assoc1 h_congr_A'
  have h_pos := eq_trans_Δ h_max h_con h_t1 h_assoc2_symm

  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))

 have h_Ax1_sx_Δ : SPA_Deduction_Internal s Δ Ax1_sx_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_sx_in
 have h_Ax1_x_Δ : SPA_Deduction_Internal s Δ Ax1_x_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_x_in
 have h_Ax_A1_1_Δ : SPA_Deduction_Internal s Δ Ax_A1_1_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax_A1_1_in
 have h_Ax_A1_2_Δ : SPA_Deduction_Internal s Δ Ax_A1_2_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax_A1_2_in
```

have h_Eq_swap_Δ : SPA_Deduction_Internal s Δ Eq_swap_forall := SPA_Deduction_Internal.hyp Δ _ h_Eq_swap_in

have h_elim4 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim3 h_Ax1_sx_Δ

have h_elim5 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim4 h_Ax1_x_Δ

have h_elim6 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim5 h_Ax_A1_1_Δ

have h_elim7 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim6 h_Ax_A1_2_Δ

have h_final_ded := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim7 h_Eq_swap_Δ

exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final_ded

Lema `zero_mul_Δ`: Producto nulo por la izquierda en Δ

Se demuestra que 0 es absorbente por la izquierda para la multiplicación dentro de Δ: para cualquier término x, el producto $0 \cdot x$ es demostrablemente igual a 0:

$$ \forall x \in \text{ArithTerm } s,\ \Delta \vdash 0 \cdot x = 0 $$

Lema auxiliar `substForm_phi_prime_zero_mul`

Define la fórmula de inducción para esta propiedad y demuestra su comportamiento bajo sustitución. La fórmula $\varphi'(v)$ es relativamente simple comparada con otras propiedades:

$$ \varphi'(v) := \text{mul_zero}(0) \to (\text{add_zero}(0) \to (\forall v\ \text{mul_succ}(0, v)) \to (0 \cdot v = 0)) $$

El lema establece que la sustitución de v por u solo afecta a la igualdad final, reemplazando v por u en $0 \cdot v = 0$. Las demás componentes (los axiomas) no contienen a v libre, por lo que permanecen inalterados.

Estructura de la demostración de `zero_mul_Δ`

1. Variable fresca: Se elige v que no ocurre en x.

2. Caso base ($v := 0$): Se demuestra $0 \cdot 0 = 0$, que es exactamente el axioma `mul_zero` para 0.

3. Paso inductivo ($v \to S(v)$): Se asume $0 \cdot v = 0$ y se debe probar $0 \cdot S(v) = 0$:

- Por `mul_succ` para 0: $0 \cdot S(v) = 0 \cdot v + 0$

- Por hipótesis de inducción: $0 \cdot v + 0 = 0 + 0$

- Por `add_zero` para 0: $0 + 0 = 0$

- Por transitividad: $0 \cdot S(v) = 0$

4. Instanciación y descarga de axiomas: La inducción proporciona $\forall v\ \varphi'(v)$. Instanciando con x, se obtiene $\varphi'(x) \in \Delta$. Las premisas se descargan con `h_mul_zero` (para `mul_zero` de 0), `h_add_zero` (para `add_zero`

de 0), y el argumento estándar de maximalidad/Henkin para $\forall v\ \text{mul_succ}(0, v)$.

Significado

`zero_mul_Δ` completa las propiedades básicas de la multiplicación en Δ, junto con `succ_mul_Δ`. Mientras que los axiomas originales solo proporcionan $x \cdot 0 = 0$ (absorbente por la derecha), este lema establece $0 \cdot x = 0$ (absorbente por la izquierda). La demostración es notablemente más simple que la de `succ_mul_Δ`, reflejando el hecho de que la absorción por cero es una propiedad más elemental que la distributividad del sucesor.

Esta propiedad, aunque aparentemente trivial, es indispensable para la manipulación de expresiones aritméticas en las pruebas de representabilidad. Por ejemplo, en `NumForm`, cuando se instancia `BetaForm_sh` con parámetros $A = 0$ o $B = 0$, pueden aparecer productos por cero que deben ser simplificados para que las igualdades con los códigos de numerales sean demostrables. La disponibilidad de `zero_mul_Δ` asegura que tales simplificaciones pueden realizarse dentro de Δ.

```
/--
  Lema auxiliar de sustitución uniforme para la multiplicación por cero por la izquierda.
-/
lemma substForm_phi_prime_zero_mul {s : Nat} (v : Nat) (u : ArithTerm s) :
  substForm v u
    (ArithForm.impl (mul_zero_axiom ArithTerm.zero)
      (ArithForm.impl (add_zero_axiom ArithTerm.zero)
        (ArithForm.impl (ArithForm.forall_form v (mul_succ_axiom ArithTerm.zero (ArithTerm.var v)))
```

```
      (ArithForm.eq (ArithTerm.mul ArithTerm.zero (ArithTerm.var v)) ArithTerm.zero)))) =

   ArithForm.impl (mul_zero_axiom ArithTerm.zero)

    (ArithForm.impl (add_zero_axiom ArithTerm.zero)

     (ArithForm.impl (ArithForm.forall_form v (mul_succ_axiom ArithTerm.zero (ArithTerm.var v)))

       (ArithForm.eq (ArithTerm.mul ArithTerm.zero u) ArithTerm.zero))) := by

  dsimp [mul_zero_axiom, add_zero_axiom, mul_succ_axiom, substForm, substTerm]

  have h_if : ∀ {α : Type} (x y : α), (if v = v then x else y) = x := fun x y => if_pos rfl

  repeat rw [h_if]


-- Producto nulo por la izquierda en Δ: 0 · x = 0

lemma zero_mul_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

   (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

   (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

   (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

   (x : ArithTerm s) :

   ArithForm.eq (ArithTerm.mul ArithTerm.zero x) ArithTerm.zero ∈ Δ := by

  let v := max_var_term x + 1

  have hvx : NotOccur v x := NotOccur_above_max_var_term x v (by omega)


  let Ax0_m0 : ArithForm s := mul_zero_axiom (ArithTerm.zero : ArithTerm s)

  let Ax0_a0 : ArithForm s := add_zero_axiom (ArithTerm.zero : ArithTerm s)

  let Ax1_m0_body : ArithForm s := mul_succ_axiom (ArithTerm.zero : ArithTerm s) (ArithTerm.var v)

  let Ax1_m0_forall : ArithForm s := ArithForm.forall_form v Ax1_m0_body

  let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.mul ArithTerm.zero (ArithTerm.var v)) ArithTerm.zero
```

```
 let φ' : ArithForm s := ArithForm.impl Ax0_m0 (ArithForm.impl Ax0_a0 (ArithForm.impl Ax1_m0_forall Eq_v))


 have h_base : SPA_Deduction_Internal s ∅ (substForm v ArithTerm.zero φ') := by
   rw [substForm_phi_prime_zero_mul v ArithTerm.zero]
   apply SPA_Deduction_Internal.imp_intro
   apply SPA_Deduction_Internal.imp_intro
   apply SPA_Deduction_Internal.imp_intro
   let Γ3 : Set (ArithForm s) := insert Ax1_m0_forall (insert Ax0_a0 (insert Ax0_m0 ∅))
   exact SPA_Deduction_Internal.hyp Γ3 Ax0_m0 (Or.inr (Or.inr (Or.inl rfl)))


 have h_step : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v (ArithForm.impl φ' (substForm v (ArithTerm.succ (ArithTerm.var v)) φ'))) := by
   apply SPA_Deduction_Internal.forall_intro
   · rw [substForm_phi_prime_zero_mul v (ArithTerm.succ (ArithTerm.var v))]
     apply SPA_Deduction_Internal.imp_intro
     apply SPA_Deduction_Internal.imp_intro
     apply SPA_Deduction_Internal.imp_intro
     apply SPA_Deduction_Internal.imp_intro
     let Γ4 : Set (ArithForm s) := insert Ax1_m0_forall (insert Ax0_a0 (insert Ax0_m0 (insert φ' ∅)))
     have h_φ' : SPA_Deduction_Internal s Γ4 φ' := SPA_Deduction_Internal.hyp Γ4 φ' (Or.inr (Or.inr (Or.inr (Or.inl rfl))))
     have h_Ax0_m0 : SPA_Deduction_Internal s Γ4 Ax0_m0 := SPA_Deduction_Internal.hyp Γ4 Ax0_m0 (Or.inr (Or.inr (Or.inl rfl)))
     have h_Ax0_a0 : SPA_Deduction_Internal s Γ4 Ax0_a0 := SPA_Deduction_Internal.hyp Γ4 Ax0_a0 (Or.inr (Or.inl rfl))
     have h_Ax1_m0 : SPA_Deduction_Internal s Γ4 Ax1_m0_forall := SPA_Deduction_Internal.hyp Γ4 Ax1_m0_forall (Or.inl rfl)


     have h_imp1 := SPA_Deduction_Internal.imp_elim Γ4 _ _ h_φ' h_Ax0_m0
```

have h_imp2 := SPA_Deduction_Internal.imp_elim Γ4 _ _ h_imp1 h_Ax0_a0

have h_Hind := SPA_Deduction_Internal.imp_elim Γ4 _ _ h_imp2 h_Ax1_m0

have h_Ax1_m0_v := SPA_Deduction_Internal.forall_elim Γ4 v Ax1_m0_body (ArithTerm.var v) h_Ax1_m0 True.intro

change SPA_Deduction_Internal s Γ4 (ArithForm.eq (ArithTerm.mul ArithTerm.zero (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.add (ArithTerm.mul ArithTerm.zero (substTerm v (ArithTerm.var v) (ArithTerm.var v))) ArithTerm.zero)) at h_Ax1_m0_v

rw [substTerm_var_same v (ArithTerm.var v)] at h_Ax1_m0_v

have h_refl_zero : SPA_Deduction_Internal s Γ4 (ArithForm.eq ArithTerm.zero ArithTerm.zero) := SPA_Deduction_Internal.eq_refl Γ4 ArithTerm.zero

have h_congr_Hind := SPA_Deduction_Internal.eq_add Γ4 _ _ _ _ h_Hind h_refl_zero

have h_t1 := SPA_Deduction_Internal.eq_trans Γ4 _ _ _ h_Ax1_m0_v h_congr_Hind

exact SPA_Deduction_Internal.eq_trans Γ4 _ _ _ h_t1 h_Ax0_a0

· intro ψ hψ

nomatch hψ

have h_ind : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ') :=

SPA_Deduction_Internal.induction ∅ φ' v h_base h_step

have h_ind_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form v φ') :=

SPA_Deduction_Internal.weaken ∅ _ Δ h_ind (Set.empty_subset _)

have h_free_phi' : IsFreeFor v x φ' := by

dsimp [φ', Ax0_m0, Ax0_a0, Ax1_m0_forall, Eq_v, mul_zero_axiom, add_zero_axiom, mul_succ_axiom, IsFreeFor]

have h1 : (if v = v then True else NotOccur v x ∧ IsFreeFor v x Ax1_m0_body) = True := if_pos rfl

```
  rw [h1]
  repeat constructor

 have h_elim := SPA_Deduction_Internal.forall_elim Δ v φ' x h_ind_Δ h_free_phi'
 rw [substForm_phi_prime_zero_mul v x] at h_elim

 have h_Ax0_m0_Δ : SPA_Deduction_Internal s Δ Ax0_m0 := SPA_Deduction_Internal.hyp Δ _ (h_mul_zero ArithTerm.zero)
 have h_Ax0_a0_Δ : SPA_Deduction_Internal s Δ Ax0_a0 := SPA_Deduction_Internal.hyp Δ _ (h_add_zero ArithTerm.zero)

 have h_elim1 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim h_Ax0_m0_Δ
 have h_elim2 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim1 h_Ax0_a0_Δ

 have h_Ax1_m0_in : Ax1_m0_forall ∈ Δ := by
  by_contra h_not
  have h_neg : ArithForm.neg Ax1_m0_forall ∈ Δ := by cases h_max Ax1_m0_forall with | inl h' => contradiction | inr h' => exact h'
  have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg Ax1_m0_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]
  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_m0_body) h_nd
  have h_subst_ax : substForm v t.val Ax1_m0_body =
    ArithForm.eq (ArithTerm.mul ArithTerm.zero (ArithTerm.succ t.val)) (ArithTerm.add (ArithTerm.mul ArithTerm.zero t.val) ArithTerm.zero) := by
    dsimp [Ax1_m0_body, mul_succ_axiom, substForm, substTerm]
    have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl
    rw [h_if]
  have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.mul ArithTerm.zero (ArithTerm.succ t.val)) (ArithTerm.add (ArithTerm.mul ArithTerm.zero t.val) ArithTerm.zero)) ∈ Δ := by
    change ArithForm.neg (substForm v t.val Ax1_m0_body) ∈ Δ at ht
```

```
    rw [h_subst_ax] at ht
    exact ht
  have h_pos := h_mul_succ ArithTerm.zero t.val
  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))

 have h_Ax1_m0_Δ : SPA_Deduction_Internal s Δ Ax1_m0_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_m0_in
 have h_final_ded := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim2 h_Ax1_m0_Δ

 exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final_ded
```

Lema `mul_comm_Δ`: Conmutatividad de la multiplicación en Δ

Se demuestra la propiedad conmutativa de la multiplicación para términos arbitrarios dentro de Δ:

$$ \forall x, y \in \text{ArithTerm } s,\ \Delta \vdash x \cdot y = y \cdot x $$

Lema auxiliar `substForm_phi_prime_mul_comm`

Define la fórmula de inducción para la conmutatividad de la multiplicación. La fórmula $\varphi'(v)$ es:

$$ \varphi'(v) := \text{mul_zero}(x) \to (0 \cdot x = 0) \to (\forall v\ \text{mul_succ}(x, v)) \to (\forall v\ S(v) \cdot x = v \cdot x + x) \to (x \cdot v = v \cdot x) $$

Las premisas incluyen:

- El axioma `mul_zero` para x

- La propiedad de absorción por cero a izquierda: $0 \cdot x = 0$ (lema `zero_mul_Δ`)

- El axioma `mul_succ` para x

- La distributividad del sucesor por la izquierda: $S(v) \cdot x = v \cdot x + x$ (lema `succ_mul_Δ`)

La conclusión es la conmutatividad $x \cdot v = v \cdot x$.

Estructura de la demostración de `mul_comm_Δ`

1. Variable fresca: Se elige v que no ocurre en x ni en y.

2. Caso base ($v := 0$): Se demuestra $x \cdot 0 = 0 \cdot x$:

- Por `mul_zero` para x: $x \cdot 0 = 0$

- Por `zero_mul_Δ` para x: $0 \cdot x = 0$

- Por transitividad y simetría: $x \cdot 0 = 0 \cdot x$

3. Paso inductivo ($v \to S(v)$): Se asume $x \cdot v = v \cdot x$ y se debe probar $x \cdot S(v) = S(v) \cdot x$:

- Por `mul_succ` para x: $x \cdot S(v) = x \cdot v + x$

- Por hipótesis de inducción: $x \cdot v + x = v \cdot x + x$

- Por `succ_mul_Δ` para v y x (invertido): $v \cdot x + x = S(v) \cdot x$

- Por transitividad: $x \cdot S(v) = S(v) \cdot x$

4. Verificación de pertenencia de axiomas cuantificados: Para $\forall v\ \text{SuccMul}(v)$ (la propiedad $S(v) \cdot x = v \cdot x + x$), se utiliza que `succ_mul_Δ` ya demuestra esta propiedad para cualquier v, por lo que su negación no puede pertenecer a Δ (argumento estándar de maximalidad y Henkin).

Significado

`mul_comm_Δ` completa las propiedades algebraicas fundamentales de la aritmética en Δ. Junto con `add_comm_Δ`, `add_assoc_Δ_term`, `zero_mul_Δ` y `succ_mul_Δ`, establece que los términos aritméticos módulo igualdad demostrable en Δ forman un semianillo conmutativo. Esta estructura algebraica es la base sobre la cual se construye toda la aritmetización de la sintaxis.

La demostración ilustra de manera particularmente clara la arquitectura en capas del desarrollo:

1. **Nivel 1:** Axiomas básicos (`add_zero`, `add_succ`, `mul_zero`, `mul_succ`)

2. **Nivel 2:** Propiedades derivadas elementales (`zero_add_Δ`, `succ_add_Δ`)

3. **Nivel 3:** Conmutatividad y asociatividad de la adición (`add_comm_Δ`, `add_assoc_Δ_term`)

4. **Nivel 4:** Propiedades de la multiplicación (`zero_mul_Δ`, `succ_mul_Δ`)

5. **Nivel 5:** Conmutatividad de la multiplicación (`mul_comm_Δ`)

Cada nivel se apoya en los anteriores, y la verificación de pertenencia de los axiomas cuantificados universalmente utiliza la maximalidad y Henkin como un mecanismo uniforme para "internalizar" propiedades que ya han sido demostradas para términos arbitrarios. Esta arquitectura modular es una de las contribuciones metodológicas más significativas de este desarrollo formal.

```
/--
  Lema auxiliar de sustitución uniforme para la conmutatividad de la multiplicación.
-/
lemma substForm_phi_prime_mul_comm {s : Nat} (x : ArithTerm s) (v : Nat)
  (hvx : NotOccur v x) (u : ArithTerm s) :
  substForm v u
   (ArithForm.impl (mul_zero_axiom x)
    (ArithForm.impl (ArithForm.eq (ArithTerm.mul ArithTerm.zero x) ArithTerm.zero)
      (ArithForm.impl (ArithForm.forall_form v (mul_succ_axiom x (ArithTerm.var v)))
       (ArithForm.impl (ArithForm.forall_form v (ArithForm.eq (ArithTerm.mul (ArithTerm.succ (ArithTerm.var v)) x) (ArithTerm.add (ArithTerm.mul (ArithTerm.var v) x) x)))
        (ArithForm.eq (ArithTerm.mul x (ArithTerm.var v)) (ArithTerm.mul (ArithTerm.var v) x)))))) =
  ArithForm.impl (mul_zero_axiom x)
   (ArithForm.impl (ArithForm.eq (ArithTerm.mul ArithTerm.zero x) ArithTerm.zero)
     (ArithForm.impl (ArithForm.forall_form v (mul_succ_axiom x (ArithTerm.var v)))
      (ArithForm.impl (ArithForm.forall_form v (ArithForm.eq (ArithTerm.mul (ArithTerm.succ (ArithTerm.var v)) x) (ArithTerm.add (ArithTerm.mul (ArithTerm.var v) x) x)))
       (ArithForm.eq (ArithTerm.mul x u) (ArithTerm.mul u x))))) := by
 dsimp [mul_zero_axiom, mul_succ_axiom, substForm, substTerm]
 rw [substTerm_fresh x v u hvx]
 have h_if : ∀ {α : Type} (x y : α), (if v = v then x else y) = x := fun x y => if_pos rfl
 repeat rw [h_if]
```

```
/--
  Conmutatividad de la Multiplicación en Δ.
  x · y = y · x
-/
lemma mul_comm_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)
  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)
  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)
  (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)
  (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)
  (x y : ArithTerm s) :
  ArithForm.eq (ArithTerm.mul x y) (ArithTerm.mul y x) ∈ Δ := by
 let v := max_var_term x + max_var_term y + 1
 have hvx : NotOccur v x := NotOccur_above_max_var_term x v (by omega)
 have hvy : NotOccur v y := NotOccur_above_max_var_term y v (by omega)

 let Ax0_x : ArithForm s := mul_zero_axiom x
 let ZeroMul_x : ArithForm s := ArithForm.eq (ArithTerm.mul ArithTerm.zero x) ArithTerm.zero
 let Ax1_x_body : ArithForm s := mul_succ_axiom x (ArithTerm.var v)
 let Ax1_x_forall : ArithForm s := ArithForm.forall_form v Ax1_x_body
 let SuccMul_body : ArithForm s := ArithForm.eq (ArithTerm.mul (ArithTerm.succ (ArithTerm.var v)) x) (ArithTerm.add (ArithTerm.mul (ArithTerm.var v) x) x)
 let SuccMul_forall : ArithForm s := ArithForm.forall_form v SuccMul_body
 let Eq_v : ArithForm s := ArithForm.eq (ArithTerm.mul x (ArithTerm.var v)) (ArithTerm.mul (ArithTerm.var v) x)

 let φ' : ArithForm s := ArithForm.impl Ax0_x (ArithForm.impl ZeroMul_x (ArithForm.impl Ax1_x_forall (ArithForm.impl SuccMul_forall Eq_v)))
```

```
  have h_base : SPA_Deduction_Internal s ∅ (substForm v ArithTerm.zero φ') := by
    rw [substForm_phi_prime_mul_comm x v hvx ArithTerm.zero]
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    apply SPA_Deduction_Internal.imp_intro
    let Γ4 : Set (ArithForm s) := insert SuccMul_forall (insert Ax1_x_forall (insert ZeroMul_x (insert Ax0_x ∅)))
    have h_Ax0_x : SPA_Deduction_Internal s Γ4 Ax0_x := SPA_Deduction_Internal.hyp Γ4 Ax0_x (Or.inr (Or.inr (Or.inr (Or.inl rfl))))
    have h_ZeroMul_x : SPA_Deduction_Internal s Γ4 ZeroMul_x := SPA_Deduction_Internal.hyp Γ4 ZeroMul_x (Or.inr (Or.inr (Or.inl rfl)))

    have h_symm_ZeroMul := SPA_Deduction_Internal.eq_symm Γ4 _ _ h_ZeroMul_x
    exact SPA_Deduction_Internal.eq_trans Γ4 _ _ _ h_Ax0_x h_symm_ZeroMul

  have h_step : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v (ArithForm.impl φ' (substForm v (ArithTerm.succ (ArithTerm.var v)) φ'))) := by
    apply SPA_Deduction_Internal.forall_intro
    · rw [substForm_phi_prime_mul_comm x v hvx (ArithTerm.succ (ArithTerm.var v))]
      apply SPA_Deduction_Internal.imp_intro
      apply SPA_Deduction_Internal.imp_intro
      apply SPA_Deduction_Internal.imp_intro
      apply SPA_Deduction_Internal.imp_intro
      apply SPA_Deduction_Internal.imp_intro
      let Γ5 : Set (ArithForm s) := insert SuccMul_forall (insert Ax1_x_forall (insert ZeroMul_x (insert Ax0_x (insert φ' ∅))))
      have h_φ' : SPA_Deduction_Internal s Γ5 φ' := SPA_Deduction_Internal.hyp Γ5 φ' (Or.inr (Or.inr (Or.inr (Or.inr (Or.inl rfl)))))
```

have h_Ax0_x : SPA_Deduction_Internal s Γ5 Ax0_x := SPA_Deduction_Internal.hyp Γ5 Ax0_x (Or.inr (Or.inr (Or.inr (Or.inl rfl))))

have h_ZeroMul_x : SPA_Deduction_Internal s Γ5 ZeroMul_x := SPA_Deduction_Internal.hyp Γ5 ZeroMul_x (Or.inr (Or.inr (Or.inl rfl)))

have h_Ax1_x : SPA_Deduction_Internal s Γ5 Ax1_x_forall := SPA_Deduction_Internal.hyp Γ5 Ax1_x_forall (Or.inr (Or.inl rfl))

have h_SuccMul : SPA_Deduction_Internal s Γ5 SuccMul_forall := SPA_Deduction_Internal.hyp Γ5 SuccMul_forall (Or.inl rfl)

have h_imp1 := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_φ' h_Ax0_x

have h_imp2 := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_imp1 h_ZeroMul_x

have h_imp3 := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_imp2 h_Ax1_x

have h_Hind := SPA_Deduction_Internal.imp_elim Γ5 _ _ h_imp3 h_SuccMul

have h_Ax1_x_v := SPA_Deduction_Internal.forall_elim Γ5 v Ax1_x_body (ArithTerm.var v) h_Ax1_x True.intro

change SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.mul (substTerm v (ArithTerm.var v) x) (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v)))) (ArithTerm.add (ArithTerm.mul (substTerm v (ArithTerm.var v) x) (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) x))) at h_Ax1_x_v

rw [substTerm_fresh x v (ArithTerm.var v) hvx, substTerm_var_same v (ArithTerm.var v)] at h_Ax1_x_v

have h_SuccMul_v := SPA_Deduction_Internal.forall_elim Γ5 v SuccMul_body (ArithTerm.var v) h_SuccMul True.intro

change SPA_Deduction_Internal s Γ5 (ArithForm.eq (ArithTerm.mul (ArithTerm.succ (substTerm v (ArithTerm.var v) (ArithTerm.var v))) (substTerm v (ArithTerm.var v) x)) (ArithTerm.add (ArithTerm.mul (substTerm v (ArithTerm.var v) (ArithTerm.var v)) (substTerm v (ArithTerm.var v) x)) (substTerm v (ArithTerm.var v) x))) at h_SuccMul_v

rw [substTerm_fresh x v (ArithTerm.var v) hvx, substTerm_var_same v (ArithTerm.var v)] at h_SuccMul_v

```
    have h_refl_x : SPA_Deduction_Internal s Γ5 (ArithForm.eq x x) := SPA_Deduction_Internal.eq_refl Γ5 x

    have h_congr_Hind := SPA_Deduction_Internal.eq_add Γ5 _ _ _ _ h_Hind h_refl_x


    have h_symm_SuccMul := SPA_Deduction_Internal.eq_symm Γ5 _ _ h_SuccMul_v


    have h_t1 := SPA_Deduction_Internal.eq_trans Γ5 _ _ _ h_Ax1_x_v h_congr_Hind

    exact SPA_Deduction_Internal.eq_trans Γ5 _ _ _ h_t1 h_symm_SuccMul

  · intro ψ hψ

    nomatch hψ


 have h_ind : SPA_Deduction_Internal s ∅ (ArithForm.forall_form v φ') :=

  SPA_Deduction_Internal.induction ∅ φ' v h_base h_step


 have h_ind_Δ : SPA_Deduction_Internal s Δ (ArithForm.forall_form v φ') :=

  SPA_Deduction_Internal.weaken ∅ _ Δ h_ind (Set.empty_subset _)


 have h_free_phi' : IsFreeFor v y φ' := by

  dsimp [φ', Ax0_x, ZeroMul_x, Ax1_x_forall, SuccMul_forall, Eq_v, mul_zero_axiom, mul_succ_axiom, IsFreeFor]

  have h1 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y Ax1_x_body) = True := if_pos rfl

  have h2 : (if v = v then True else NotOccur v y ∧ IsFreeFor v y SuccMul_body) = True := if_pos rfl

  rw [h1, h2]

  repeat constructor


 have h_elim := SPA_Deduction_Internal.forall_elim Δ v φ' y h_ind_Δ h_free_phi'

 rw [substForm_phi_prime_mul_comm x v hvx y] at h_elim
```

```
  have h_Ax0_x_Δ : SPA_Deduction_Internal s Δ Ax0_x := SPA_Deduction_Internal.hyp Δ
_ (h_mul_zero x)

  have h_ZeroMul_x_in := zero_mul_Δ Δ h_max h_con h_henkin h_add_zero h_mul_zero
h_mul_succ x

  have h_ZeroMul_x_Δ : SPA_Deduction_Internal s Δ ZeroMul_x :=
SPA_Deduction_Internal.hyp Δ _ h_ZeroMul_x_in


  have h_elim1 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim h_Ax0_x_Δ

  have h_elim2 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim1 h_ZeroMul_x_Δ


  have h_Ax1_x_in : Ax1_x_forall ∈ Δ := by

    by_contra h_not

    have h_neg : ArithForm.neg Ax1_x_forall ∈ Δ := by cases h_max Ax1_x_forall with | inl
h' => contradiction | inr h' => exact h'

    have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg
Ax1_x_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]

    obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg Ax1_x_body) h_nd

    have h_subst_ax : substForm v t.val Ax1_x_body =

      ArithForm.eq (ArithTerm.mul x (ArithTerm.succ t.val)) (ArithTerm.add
(ArithTerm.mul x t.val) x) := by

      dsimp [Ax1_x_body, mul_succ_axiom, substForm, substTerm]

      rw [substTerm_fresh x v t.val hvx]

      have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl

      rw [h_if]

    have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.mul x (ArithTerm.succ t.val))
(ArithTerm.add (ArithTerm.mul x t.val) x)) ∈ Δ := by

      change ArithForm.neg (substForm v t.val Ax1_x_body) ∈ Δ at ht

      rw [h_subst_ax] at ht

      exact ht

    have h_pos := h_mul_succ x t.val
```

```
  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))


 have h_SuccMul_in : SuccMul_forall ∈ Δ := by

  by_contra h_not

  have h_neg : ArithForm.neg SuccMul_forall ∈ Δ := by cases h_max SuccMul_forall with | inl h' => contradiction | inr h' => exact h'

  have h_nd : ArithForm.neg (ArithForm.forall_form v (ArithForm.neg (ArithForm.neg SuccMul_body))) ∈ Δ := by rwa [← neg_forall_iff_neg_forall_double_neg h_max h_con v]

  obtain ⟨t, ht⟩ := h_henkin v (ArithForm.neg SuccMul_body) h_nd

  have h_subst_ax : substForm v t.val SuccMul_body =

   ArithForm.eq (ArithTerm.mul (ArithTerm.succ t.val) x) (ArithTerm.add (ArithTerm.mul t.val x) x) := by

   dsimp [SuccMul_body, substForm, substTerm]

   rw [substTerm_fresh x v t.val hvx]

   have h_if : (if v = v then t.val else ArithTerm.var v) = t.val := if_pos rfl

   rw [h_if]

  have ht' : ArithForm.neg (ArithForm.eq (ArithTerm.mul (ArithTerm.succ t.val) x) (ArithTerm.add (ArithTerm.mul t.val x) x)) ∈ Δ := by

   change ArithForm.neg (substForm v t.val SuccMul_body) ∈ Δ at ht

   rw [h_subst_ax] at ht

   exact ht

  have h_pos := succ_mul_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ t.val x

  exact h_con (SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_pos) (SPA_Deduction_Internal.hyp Δ _ ht'))


 have h_Ax1_x_Δ : SPA_Deduction_Internal s Δ Ax1_x_forall := SPA_Deduction_Internal.hyp Δ _ h_Ax1_x_in

 have h_SuccMul_Δ : SPA_Deduction_Internal s Δ SuccMul_forall := SPA_Deduction_Internal.hyp Δ _ h_SuccMul_in
```

```
  have h_elim3 := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim2 h_Ax1_x_Δ
  have h_final_ded := SPA_Deduction_Internal.imp_elim Δ _ _ h_elim3 h_SuccMul_Δ

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final_ded
```

Lemas de manipulación algebraica avanzada y unicidad del resto euclídeo

Se demuestran cuatro lemas que extienden significativamente las capacidades algebraicas dentro de Δ, culminando con el teorema de unicidad del resto euclídeo para numerales. Estos resultados son esenciales para la demostración de funcionalidad de `NumForm`.

Lema `add_cancel_M_right_Δ`: Cancelación de un término común en sumas agrupadas

Establece una variante de la cancelación aditiva donde el término a cancelar está "en el medio" de sumas asociadas:

$$ \Delta \vdash (A + M) + r_1 = (B + M) + r_2 \implies \Delta \vdash A + r_1 = B + r_2 $$

La demostración utiliza una serie de reordenamientos basados en asociatividad (`add_assoc_Δ_term`) y conmutatividad (`add_comm_Δ`) para transformar ambos

lados de la igualdad en la forma $(A + r_1) + M = (B + r_2) + M$, y luego aplica `add_cancel_right_Δ` para cancelar M.

Lema `succ_mul_reassoc_Δ`: Reasociación del sucesor en la multiplicación con adición

Establece una identidad algebraica que relaciona la multiplicación por un sucesor con sumas:

$$ \Delta \vdash S(q) \cdot M + r = (q \cdot M + r) + M $$

La demostración utiliza `succ_mul_Δ` para expandir $S(q) \cdot M = q \cdot M + M$, y luego reordena la suma $(q \cdot M + M) + r$ usando asociatividad y conmutatividad para obtener $(q \cdot M + r) + M$.

Lema `zero_mul_add_Δ`: Simplificación de $0 \cdot m + k$

$$ \Delta \vdash 0 \cdot m + k = k $$

Consecuencia directa de `zero_mul_Δ` (que da $0 \cdot m = 0$) y `zero_add_Δ` (que da $0 + k = k$).

Lema `succ_mul_add_hom_Δ`: Homomorfismo para $(n'+1) \cdot m + k$

Establece que el término compuesto $\overline{n'+1} \cdot \overline{m} + \overline{k}$ es demostrablemente igual al numeral $\overline{(n'+1) \cdot m + k}$:

$$ \Delta \vdash \overline{n'+1} \cdot \overline{m} + \overline{k} = \overline{(n'+1) \cdot m + k} $$

La demostración utiliza los homomorfismos de la multiplicación y la adición.

Lema `euclidean_remainder_unique_numeral_Δ`: Unicidad del resto euclídeo para numerales

Este es el resultado culminante de la sección. Establece que si dos igualdades de la forma $n_1 \cdot m + k_1 = n_2 \cdot m + k_2$ son demostrables en Δ con $k_1, k_2 < m$, entonces necesariamente $k_1 = k_2$ (en el metalenguaje):

$$ \forall n_1, n_2, m, k_1, k_2 \in \mathbb{N},\ k_1 < m \land k_2 < m \land \Delta \vdash \overline{n_1} \cdot \overline{m} + \overline{k_1} = \overline{n_2} \cdot \overline{m} + \overline{k_2} \implies k_1 = k_2 $$

Estructura de la demostración

La prueba procede por inducción principal sobre n_1 con generalización de n_2, seguida de un análisis de casos sobre n_2:

Caso $n_1 = 0$:

- Si $n_2 = 0$: Ambos lados se simplifican a $\overline{k_1}$ y $\overline{k_2}$ respectivamente. Si $k_1 \neq k_2$, los numerales correspondientes son distintos y `prove_distinct_numerals_Δ` da una contradicción con la consistencia.

- Si $n_2 = n_2' + 1$: El lado derecho es al menos $\overline{m}$ (porque $(n_2'+1) \cdot m \geq m$), mientras que el lado izquierdo es $\overline{k_1}$ con $k_1 < m$. La igualdad de numerales forzaría $k_1 = (n_2'+1) \cdot m + k_2 \geq m$, contradicción.

Caso $n_1 = n_1' + 1$:

- Si $n_2 = 0$: Simétrico al caso anterior.

- Si $n_2 = n_2' + 1$: Se aplica `succ_mul_reassoc_Δ` a ambos lados para obtener $(n_1' \cdot m + k_1) + m = (n_2' \cdot m + k_2) + m \in \Delta$. Por `add_cancel_right_Δ`, se cancelan las m y se obtiene $n_1' \cdot m + k_1 = n_2' \cdot m + k_2 \in \Delta$. La hipótesis de inducción aplicada a n_1' y n_2' concluye $k_1 = k_2$.

Significado

`euclidean_remainder_unique_numeral_Δ` es la contraparte interna del algoritmo de la división euclídea. Establece que la representación de un número como $n \cdot m + k$ con $k < m$ es única, una propiedad fundamental de la aritmética. Este lema es el ingrediente final necesario para la demostración de funcionalidad de `NumForm`, donde será utilizado para garantizar que el código del numeral es único: si dos secuencias de códigos satisfacen las condiciones de `NumForm`, entonces producen exactamente el mismo código. La demostración utiliza inducción en el metalenguaje sobre los cocientes n_1 y n_2, reduciendo progresivamente el problema hasta alcanzar el caso base donde la unicidad se sigue de la distinción de numerales.

-- Reordenamiento y Cancelación de M por la Derecha en Δ: (A + M) + r1 = (B + M) + r2 → A + r1 = B + r2

lemma add_cancel_M_right_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(A B M r1 r2 : TermClosed s)

(h_eq : ArithForm.eq (ArithTerm.add (ArithTerm.add A.val M.val) r1.val)

(ArithTerm.add (ArithTerm.add B.val M.val) r2.val) ∈ Δ) :

ArithForm.eq (ArithTerm.add A.val r1.val) (ArithTerm.add B.val r2.val) ∈ Δ := by

have h_assoc_L := add_assoc_Δ_term Δ h_max h_con h_henkin h_add_zero h_add_succ A.val M.val r1.val

have h_comm_Mr1 := add_comm_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ M.val r1.val

have h_congr_L := eq_add_Δ h_max h_con (eq_refl_in_Δ Δ h_max h_con A.val) h_comm_Mr1

have h_assoc_L2 := add_assoc_Δ_term Δ h_max h_con h_henkin h_add_zero h_add_succ A.val r1.val M.val

have h_assoc_L2_symm := eq_symm_Δ h_max h_con h_assoc_L2

have h_t1_L := eq_trans_Δ h_max h_con h_assoc_L h_congr_L

have h_L_reorder := eq_trans_Δ h_max h_con h_t1_L h_assoc_L2_symm

have h_assoc_R := add_assoc_Δ_term Δ h_max h_con h_henkin h_add_zero h_add_succ B.val M.val r2.val

have h_comm_Mr2 := add_comm_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ M.val r2.val

have h_congr_R := eq_add_Δ h_max h_con (eq_refl_in_Δ Δ h_max h_con B.val) h_comm_Mr2

have h_assoc_R2 := add_assoc_Δ_term Δ h_max h_con h_henkin h_add_zero h_add_succ B.val r2.val M.val

have h_assoc_R2_symm := eq_symm_Δ h_max h_con h_assoc_R2

have h_t1_R := eq_trans_Δ h_max h_con h_assoc_R h_congr_R

have h_R_reorder := eq_trans_Δ h_max h_con h_t1_R h_assoc_R2_symm

```
  have h_L_symm := eq_symm_Δ h_max h_con h_L_reorder
  have h_mid := eq_trans_Δ h_max h_con h_L_symm h_eq
  have h_final := eq_trans_Δ h_max h_con h_mid h_R_reorder

  let Ar1 : TermClosed s := ⟨ArithTerm.add A.val r1.val, by
    dsimp [isClosed]
    rw [A.property, r1.property]
    rfl⟩
  let Br2 : TermClosed s := ⟨ArithTerm.add B.val r2.val, by
    dsimp [isClosed]
    rw [B.property, r2.property]
    rfl⟩

  exact add_cancel_right_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ Ar1 Br2 M
h_final



-- Reasociación de la Multiplicación por Sucesor con la Adición en Δ:
-- S(q) · M + r = (q · M + r) + M
lemma succ_mul_reassoc_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ) (h_henkin : HenkinSet Δ)
    (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)
    (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)
    (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)
    (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)
    (q M r : ArithTerm s) :
    ArithForm.eq (ArithTerm.add (ArithTerm.mul (ArithTerm.succ q) M) r)
            (ArithTerm.add (ArithTerm.add (ArithTerm.mul q M) r) M) ∈ Δ := by
```

have h_sm := succ_mul_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ q M

have h_refl_r := eq_refl_in_Δ Δ h_max h_con r

have h1 := eq_add_Δ h_max h_con h_sm h_refl_r

have h2 := add_assoc_Δ_term Δ h_max h_con h_henkin h_add_zero h_add_succ (ArithTerm.mul q M) M r

have h_comm := add_comm_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ M r

have h_refl_qM := eq_refl_in_Δ Δ h_max h_con (ArithTerm.mul q M)

have h3 := eq_add_Δ h_max h_con h_refl_qM h_comm

have h4 := add_assoc_Δ_term Δ h_max h_con h_henkin h_add_zero h_add_succ (ArithTerm.mul q M) r M

have h4_symm := eq_symm_Δ h_max h_con h4

have h_t1 := eq_trans_Δ h_max h_con h1 h2

have h_t2 := eq_trans_Δ h_max h_con h_t1 h3

exact eq_trans_Δ h_max h_con h_t2 h4_symm

-- Simplificación de 0 · m + k = k en Δ

lemma zero_mul_add_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

(h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

(m k : ArithTerm s) :

ArithForm.eq (ArithTerm.add (ArithTerm.mul ArithTerm.zero m) k) k ∈ Δ := by

have h_zm := zero_mul_Δ Δ h_max h_con h_henkin h_add_zero h_mul_zero h_mul_succ m

```
  have h_refl_k := eq_refl_in_Δ Δ h_max h_con k
  have h_congr := eq_add_Δ h_max h_con h_zm h_refl_k
  have h_za := zero_add_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ k
  exact eq_trans_Δ h_max h_con h_congr h_za

-- Homomorfismo numérico para (n' + 1) · m + k
lemma succ_mul_add_hom_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ)
(h_con : Consistent Δ)
    (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)
    (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)
    (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)
    (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)
    (n' m k : Nat) :
    let V := (n' + 1) * m + k
    ArithForm.eq (ArithTerm.add (ArithTerm.mul (natToTerm (s := s) (n' + 1))
(natToTerm (s := s) m)) (natToTerm (s := s) k))
            (natToTerm (s := s) V) ∈ Δ := by
  intro V
  have h_mul := natToTerm_mul_hom Δ h_max h_con h_add_zero h_add_succ
h_mul_zero h_mul_succ (n' + 1) m
  have h_mul_symm := eq_symm_Δ h_max h_con h_mul
  have h_refl_k := eq_refl_in_Δ Δ h_max h_con (natToTerm (s := s) k)
  have h_congr := eq_add_Δ h_max h_con h_mul_symm h_refl_k
  have h_add := natToTerm_add_hom Δ h_max h_con h_add_zero h_add_succ ((n' + 1) *
m) k
  have h_add_symm := eq_symm_Δ h_max h_con h_add
  exact eq_trans_Δ h_max h_con h_congr h_add_symm

/--
    Unicidad del Resto Euclídeo sobre Numerales en Δ.
```

Si $n_1 \cdot m + k_1 = n_2 \cdot m + k_2$ con $k_1 < m$ y $k_2 < m$, entonces $k_1 = k_2$.

-/

lemma euclidean_remainder_unique_numeral_Δ {s : Nat} (Δ : Set (ArithForm s)) (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)

(h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)

(h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)

(h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)

(h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)

(n1 n2 m k1 k2 : Nat) (hk1 : k1 < m) (hk2 : k2 < m)

(h_eq : ArithForm.eq (ArithTerm.add (ArithTerm.mul (natToTerm (s := s) n1) (natToTerm (s := s) m)) (natToTerm (s := s) k1))

(ArithTerm.add (ArithTerm.mul (natToTerm (s := s) n2) (natToTerm (s := s) m)) (natToTerm (s := s) k2)) ∈ Δ) :

k1 = k2 := by

induction n1 generalizing n2 with

| zero =>

have h_LHS_simpl := zero_mul_add_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ (natToTerm (s := s) m) (natToTerm (s := s) k1)

have h_LHS_symm := eq_symm_Δ h_max h_con h_LHS_simpl

cases n2 with

| zero =>

have h_RHS_simpl := zero_mul_add_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ (natToTerm (s := s) m) (natToTerm (s := s) k2)

have h_t1 := eq_trans_Δ h_max h_con h_LHS_symm h_eq

have h_k1_k2 := eq_trans_Δ h_max h_con h_t1 h_RHS_simpl

by_contra h_neq

have h_neg := prove_distinct_numerals_Δ h_max h_con k1 k2 h_neq

have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=

SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_k1_k2) (SPA_Deduction_Internal.hyp Δ _ h_neg)

```
    exact False.elim (h_con h_bot)
  | succ n2' =>
    have h_RHS_hom := succ_mul_add_hom_Δ Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ n2' m k2
    have h_t1 := eq_trans_Δ h_max h_con h_LHS_symm h_eq
    have h_k1_V := eq_trans_Δ h_max h_con h_t1 h_RHS_hom
    let V := (n2' + 1) * m + k2
    have h_V_ge : V ≥ m := by
      dsimp [V]
      have : (n2' + 1) * m = m + n2' * m := by ring
      omega
    have h_neq : k1 ≠ V := by omega
    have h_neg := prove_distinct_numerals_Δ h_max h_con k1 V h_neq
    have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
      SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_k1_V) (SPA_Deduction_Internal.hyp Δ _ h_neg)
    exact False.elim (h_con h_bot)

 | succ n1' ih =>
  cases n2 with
  | zero =>
    have h_RHS_simpl := zero_mul_add_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ (natToTerm (s := s) m) (natToTerm (s := s) k2)
    have h_LHS_hom := succ_mul_add_hom_Δ Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ n1' m k1
    have h_LHS_symm := eq_symm_Δ h_max h_con h_LHS_hom
    have h_t1 := eq_trans_Δ h_max h_con h_LHS_symm h_eq
    have h_V_k2 := eq_trans_Δ h_max h_con h_t1 h_RHS_simpl
    let U := (n1' + 1) * m + k1
    have h_U_ge : U ≥ m := by
```

```
      dsimp [U]
      have : (n1' + 1) * m = m + n1' * m := by ring
      omega
    have h_neq : U ≠ k2 := by omega
    have h_neg := prove_distinct_numerals_Δ h_max h_con U k2 h_neq
    have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=
      SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_V_k2) (SPA_Deduction_Internal.hyp Δ _ h_neg)
    exact False.elim (h_con h_bot)
  | succ n2' =>
    have h_LHS_reassoc := succ_mul_reassoc_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ (natToTerm (s := s) n1') (natToTerm (s := s) m) (natToTerm (s := s) k1)
    have h_RHS_reassoc := succ_mul_reassoc_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ (natToTerm (s := s) n2') (natToTerm (s := s) m) (natToTerm (s := s) k2)
    have h_LHS_symm := eq_symm_Δ h_max h_con h_LHS_reassoc
    have h_t1 := eq_trans_Δ h_max h_con h_LHS_symm h_eq
    have h_reassoc_eq := eq_trans_Δ h_max h_con h_t1 h_RHS_reassoc

    let A : TermClosed s := ⟨ArithTerm.add (ArithTerm.mul (natToTerm (s := s) n1') (natToTerm (s := s) m)) (natToTerm (s := s) k1), by
      dsimp [isClosed]
      rw [isClosed_natToTerm, isClosed_natToTerm, isClosed_natToTerm]
      rfl⟩
    let B : TermClosed s := ⟨ArithTerm.add (ArithTerm.mul (natToTerm (s := s) n2') (natToTerm (s := s) m)) (natToTerm (s := s) k2), by
      dsimp [isClosed]
      rw [isClosed_natToTerm, isClosed_natToTerm, isClosed_natToTerm]
      rfl⟩
    let M_c : TermClosed s := ⟨natToTerm (s := s) m, isClosed_natToTerm m⟩
```

```
    have h_cancel := add_cancel_right_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ A B M_c h_reassoc_eq
    exact ih n2' h_cancel
```

Lemas de irreflexividad del orden y unicidad del resto euclídeo generalizado

Se demuestran dos lemas que constituyen los últimos ingredientes técnicos para la prueba de funcionalidad de `NumForm`. El primero establece que ningún término cerrado puede ser estrictamente menor que sí mismo en Δ. El segundo generaliza la unicidad del resto euclídeo desde numerales a términos cerrados arbitrarios.

Lema `lt_irrefl_closed`: Irreflexividad de $<$ para términos cerrados

Establece que para ningún término cerrado x y ninguna variable testigo w puede ocurrir $x < x$ en Δ:

$$ \forall x \in \text{TermClosed } s,\ \forall w \in \mathbb{N},\ \Delta \vdash x < x \implies \bot $$

Estructura de la demostración

1. **Descomposición de la hipótesis:** $x < x$ significa $\exists v_w\ (x = x + S(v_w)) \in \Delta$. Por la propiedad de Henkin, existe un testigo cerrado d tal que $x = x + S(d) \in \Delta$.

2. **Manipulación algebraica:** Se utiliza $0 + x = x$ (lema `zero_add_Δ`) para reescribir la igualdad como $0 + x = x + S(d)$. Por conmutatividad de la adición (`add_comm_Δ`), $x + S(d) = S(d) + x$.

3. **Cancelación:** Se aplica `add_cancel_right_Δ` a la igualdad $0 + x = S(d) + x$ para obtener $0 = S(d) \in \Delta$.

4. **Contradicción con Peano:** El primer axioma de Peano (`ax_peano1`) establece $\neg(S(d) = 0) \in \Delta$. La presencia simultánea de $0 = S(d)$ y su negación contradice la consistencia de Δ.

Lema `euclid_remainder_unique_closed`: Unicidad del resto euclídeo para términos cerrados

Generaliza `euclidean_remainder_unique_numeral_Δ` a términos cerrados arbitrarios que sean demostrablemente iguales a numerales. Si dos representaciones $q_1 \cdot M + r_1 = q_2 \cdot M + r_2$ son demostrables en Δ, con $r_1 < M$ y $r_2 < M$, y además M, q_1, q_2 son demostrablemente iguales a numerales, entonces $r_1 = r_2 \in \Delta$:

$$ \Delta \vdash q_1 \cdot M + r_1 = q_2 \cdot M + r_2 \land r_1 < M \land r_2 < M \implies \Delta \vdash r_1 = r_2 $$

bajo las hipótesis adicionales de que $M = \overline{k_m}$, $q_1 = \overline{n_1}$, $q_2 = \overline{n_2}$ para algunos numerales.

Estructura de la demostración

La prueba consta de seis etapas:

1. Extracción de parámetros numerales: Se obtienen k_m, n_1, n_2 tales que $M = \overline{k_m}$, $q_1 = \overline{n_1}$, $q_2 = \overline{n_2}$ en Δ.

2. Determinación de r_1 como numeral: De $r_1 < M$ y $M = \overline{k_m}$, se obtiene $r_1 < \overline{k_m}$. Por `lt_numeral_bounded`, existe $k_1 < k_m$ tal que $r_1 = \overline{k_1} \in \Delta$. El argumento requiere extraer el testigo de la fórmula de orden usando Henkin, reescribir con $M = \overline{k_m}$, y aplicar `lt_numeral_bounded`.

3. Determinación de r_2 como numeral: Análogo al paso anterior, obteniendo $k_2 < k_m$ con $r_2 = \overline{k_2} \in \Delta$.

4. Reducción a numerales: Se sustituyen q_1, q_2, M, r_1, r_2 por sus numerales correspondientes en la igualdad $q_1 \cdot M + r_1 = q_2 \cdot M + r_2$, usando congruencia de la multiplicación y la adición. Se obtiene:

$$ \overline{n_1} \cdot \overline{k_m} + \overline{k_1} = \overline{n_2} \cdot \overline{k_m} + \overline{k_2} \in \Delta $$

5. Aplicación del lema numeral: `euclidean_remainder_unique_numeral_Δ` implica $k_1 = k_2$ en el metalenguaje.

6. Conclusión: De $k_1 = k_2$, se tiene $\overline{k_1} = \overline{k_2}$. Por transitividad con $r_1 = \overline{k_1}$ y $r_2 = \overline{k_2}$ (esta última por simetría), se obtiene $r_1 = r_2 \in \Delta$.

Significado conjunto

Estos dos lemas proporcionan las herramientas finales para la demostración de funcionalidad de `NumForm`. La irreflexividad de $<$ garantiza que la condición de orden $k < w$ en la definición de `NumForm` es estricta, evitando ciclos en la construcción inductiva. La unicidad del resto euclídeo generalizado garantiza que si dos secuencias de códigos satisfacen las condiciones de `NumForm` para el mismo n, entonces producen exactamente el mismo código z.

La combinación de `lt_irrefl_closed` y `euclid_remainder_unique_closed` con los resultados de representabilidad previos (`beta_sh_representable`, `IsSuccCode_representable`) y los lemas de manipulación algebraica (`add_cancel_right_Δ`, `succ_mul_reassoc_Δ`, etc.) completa el aparato técnico necesario para establecer que `NumForm` es una fórmula representadora en el sentido de `IsRepresentable`. La arquitectura de estas demostraciones —reducción de términos cerrados a numerales mediante `lt_numeral_bounded`, manipulación algebraica interna, y aplicación de unicidad en el metalenguaje— es un patrón que caracteriza el enfoque sintáctico de la aritmetización en el sistema SPA.

```
/--
  Paso 1.1: Lema de irreflexividad de `<` para términos cerrados.
  Demuestra que ningún término cerrado x satisface x < x en Δ.
-/
lemma lt_irrefl_closed {s : Nat} (Δ : Set (ArithForm s))
  (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)
```

```
  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)
  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)
  (x : TermClosed s) (w : Nat)
  (h_lt : lt_form x.val x.val w ∈ Δ) : False := by
 -- 1. Descomponer lt_form x.val x.val w como existencia en Δ
 have h_ex : exists_form w (ArithForm.eq x.val (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var w)))) ∈ Δ := h_lt

 -- 2. Extraer el testigo Henkin d
 obtain ⟨d, hd⟩ := h_henkin w (ArithForm.eq x.val (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var w)))) h_ex

 -- 3. Simplificar la sustitución del testigo sobre términos cerrados
 have hd_simp : substForm w d.val (ArithForm.eq x.val (ArithTerm.add x.val (ArithTerm.succ (ArithTerm.var w)))) =
       ArithForm.eq x.val (ArithTerm.add x.val (ArithTerm.succ d.val)) := by
  dsimp [substForm, substTerm]
  have h_x_fresh := substTerm_fresh x.val w d.val (NotOccur_of_isClosed x.val x.property w)
  have h_if : (if w = w then d.val else ArithTerm.var w) = d.val := if_pos rfl
  rw [h_x_fresh, h_if]
 rw [hd_simp] at hd
 -- hd : ArithForm.eq x.val (ArithTerm.add x.val (ArithTerm.succ d.val)) ∈ Δ

 -- 4. Obtener las propiedades algebraicas de 0 y + en Δ
 have h_za := zero_add_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ x.val
 -- h_za : ArithForm.eq (ArithTerm.add ArithTerm.zero x.val) x.val ∈ Δ

 have h_comm := add_comm_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ x.val (ArithTerm.succ d.val)
```

-- h_comm : ArithForm.eq (ArithTerm.add x.val (ArithTerm.succ d.val)) (ArithTerm.add (ArithTerm.succ d.val) x.val) ∈ Δ

-- 5. Encadenar transitividades: (0 + x) = (S(d) + x)

have h_t1 := eq_trans_Δ h_max h_con h_za hd

have h_t2 := eq_trans_Δ h_max h_con h_t1 h_comm

-- h_t2 : ArithForm.eq (ArithTerm.add ArithTerm.zero x.val) (ArithTerm.add (ArithTerm.succ d.val) x.val) ∈ Δ

-- 6. Cancelar x por la derecha utilizando add_cancel_right_Δ

let zero_c : TermClosed s := zero_closed

let succ_d_c : TermClosed s := succ_closed d

have h_cancel := add_cancel_right_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ zero_c succ_d_c x h_t2

-- h_cancel : ArithForm.eq ArithTerm.zero (ArithTerm.succ d.val) ∈ Δ

-- 7. Derivar la contradicción con el Axioma 1 de Peano

have h_cancel_symm := eq_symm_Δ h_max h_con h_cancel

-- h_cancel_symm : ArithForm.eq (ArithTerm.succ d.val) ArithTerm.zero ∈ Δ

have h_peano1 := SPA_Deduction_Internal.ax_peano1 Δ d.val

have h_peano1_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_peano1

have h_bot : SPA_Deduction_Internal s Δ ArithForm.bottom :=

SPA_Deduction_Internal.neg_elim Δ _ (SPA_Deduction_Internal.hyp Δ _ h_cancel_symm) (SPA_Deduction_Internal.hyp Δ _ h_peano1_in)

exact h_con h_bot

```
/--
   Paso 1.2: Lema de unicidad del resto euclídeo generalizado para términos cerrados.
   Demuestra que si dos representaciones euclídeas q₁·M + r₁ y q₂·M + r₂ son iguales en Δ,
   con r₁ < M y r₂ < M, entonces sus restos r₁ y r₂ son iguales en Δ.
-/
lemma euclid_remainder_unique_closed {s : Nat} (Δ : Set (ArithForm s))
   (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)
   (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)
   (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)
   (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)
   (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)
   (q1 q2 r1 r2 M : TermClosed s)
   (h_M_pos : ∃ k, k ≥ 1 ∧ ArithForm.eq M.val (natToTerm k) ∈ Δ)
   (hq1 : ∃ n1, ArithForm.eq q1.val (natToTerm n1) ∈ Δ)
   (hq2 : ∃ n2, ArithForm.eq q2.val (natToTerm n2) ∈ Δ)
   (h_eq : ArithForm.eq (ArithTerm.add (ArithTerm.mul q1.val M.val) r1.val)
                        (ArithTerm.add (ArithTerm.mul q2.val M.val) r2.val) ∈ Δ)
   (h_lt1 : lt_form r1.val M.val 5 ∈ Δ)
   (h_lt2 : lt_form r2.val M.val 5 ∈ Δ) :
   ArithForm.eq r1.val r2.val ∈ Δ := by
  -- 1. Extraer k_m, n1, n2 de las hipótesis de representación numeral
  obtain ⟨k_m, hk_m_pos, h_M_eq⟩ := h_M_pos
  obtain ⟨n1, h_q1_eq⟩ := hq1
  obtain ⟨n2, h_q2_eq⟩ := hq2
```

-- 2. Demostrar que r_1 es igual en Δ a un numeral $k_1 < k_m$

obtain ⟨d1, hd1⟩ := h_henkin 5 (ArithForm.eq M.val (ArithTerm.add r1.val (ArithTerm.succ (ArithTerm.var 5)))) h_lt1

have hd1_simp : substForm 5 d1.val (ArithForm.eq M.val (ArithTerm.add r1.val (ArithTerm.succ (ArithTerm.var 5)))) =

ArithForm.eq M.val (ArithTerm.add r1.val (ArithTerm.succ d1.val)) := by

dsimp [substForm, substTerm]

rw [substTerm_fresh M.val 5 d1.val (NotOccur_of_isClosed M.val M.property 5),

substTerm_fresh r1.val 5 d1.val (NotOccur_of_isClosed r1.val r1.property 5)]

rw [hd1_simp] at hd1

have h_M_symm := eq_symm_Δ h_max h_con h_M_eq

have h_m_d1 := eq_trans_Δ h_max h_con h_M_symm hd1

have h_subst1 : substForm 1 d1.val (ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r1.val (ArithTerm.succ (ArithTerm.var 1)))) =

ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r1.val (ArithTerm.succ d1.val)) := by

dsimp [substForm, substTerm]

rw [substTerm_fresh (natToTerm (s := s) k_m) 1 d1.val (NotOccur_of_isClosed _ (isClosed_natToTerm k_m) 1),

substTerm_fresh r1.val 1 d1.val (NotOccur_of_isClosed r1.val r1.property 1)]

have h_subst1_in : substForm 1 d1.val (ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r1.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ := by

rwa [h_subst1]

have hfree1 : IsFreeFor 1 d1.val (ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r1.val (ArithTerm.succ (ArithTerm.var 1)))) := IsFreeFor_closed _ 1 d1

have hex1 := exists_intro Δ 1 _ d1.val hfree1 (SPA_Deduction_Internal.hyp Δ _ h_subst1_in)

have hex1_in : exists_form 1 (ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r1.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ :=

maximal_consistent_closed_under_deduction Δ h_max h_con _ hex1

have h_r1_lt_m : substForm 0 r1.val (lt_form (ArithTerm.var 0) (natToTerm k_m) 1) ∈ Δ := by

rw [substForm_lt_form r1 (natToTerm k_m) (isClosed_natToTerm k_m)]

exact hex1_in

obtain ⟨k1, hk1_lt, h_r1_eq⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ k_m r1 h_r1_lt_m

-- 3. Demostrar analógicamente que r_2 es igual en Δ a un numeral $k_2 < k_m$

obtain ⟨d2, hd2⟩ := h_henkin 5 (ArithForm.eq M.val (ArithTerm.add r2.val (ArithTerm.succ (ArithTerm.var 5)))) h_lt2

have hd2_simp : substForm 5 d2.val (ArithForm.eq M.val (ArithTerm.add r2.val (ArithTerm.succ (ArithTerm.var 5)))) =

ArithForm.eq M.val (ArithTerm.add r2.val (ArithTerm.succ d2.val)) := by

dsimp [substForm, substTerm]

rw [substTerm_fresh M.val 5 d2.val (NotOccur_of_isClosed M.val M.property 5),

substTerm_fresh r2.val 5 d2.val (NotOccur_of_isClosed r2.val r2.property 5)]

rw [hd2_simp] at hd2

have h_m_d2 := eq_trans_Δ h_max h_con h_M_symm hd2

have h_subst2 : substForm 1 d2.val (ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r2.val (ArithTerm.succ (ArithTerm.var 1)))) =

ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r2.val (ArithTerm.succ d2.val)) := by

dsimp [substForm, substTerm]

rw [substTerm_fresh (natToTerm (s := s) k_m) 1 d2.val (NotOccur_of_isClosed _ (isClosed_natToTerm k_m) 1),

substTerm_fresh r2.val 1 d2.val (NotOccur_of_isClosed r2.val r2.property 1)]

have h_subst2_in : substForm 1 d2.val (ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r2.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ := by

rwa [h_subst2]

have hfree2 : IsFreeFor 1 d2.val (ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r2.val (ArithTerm.succ (ArithTerm.var 1)))) := IsFreeFor_closed _ 1 d2

have hex2 := exists_intro Δ 1 _ d2.val hfree2 (SPA_Deduction_Internal.hyp Δ _ h_subst2_in)

have hex2_in : exists_form 1 (ArithForm.eq (natToTerm (s := s) k_m) (ArithTerm.add r2.val (ArithTerm.succ (ArithTerm.var 1)))) ∈ Δ :=

maximal_consistent_closed_under_deduction Δ h_max h_con _ hex2

have h_r2_lt_m : substForm 0 r2.val (lt_form (ArithTerm.var 0) (natToTerm k_m) 1) ∈ Δ := by

rw [substForm_lt_form r2 (natToTerm k_m) (isClosed_natToTerm k_m)]

exact hex2_in

obtain ⟨k2, hk2_lt, h_r2_eq⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ k_m r2 h_r2_lt_m

-- 4. Sustituir todas las variables por sus numerales correspondientes en h_eq

have h_mul1 := eq_mul_Δ h_max h_con h_q1_eq h_M_eq

have h_lhs_congr := eq_add_Δ h_max h_con h_mul1 h_r1_eq

have h_mul2 := eq_mul_Δ h_max h_con h_q2_eq h_M_eq

have h_rhs_congr := eq_add_Δ h_max h_con h_mul2 h_r2_eq

have h_lhs_symm := eq_symm_Δ h_max h_con h_lhs_congr

have h_t1 := eq_trans_Δ h_max h_con h_lhs_symm h_eq

have h_num_eq := eq_trans_Δ h_max h_con h_t1 h_rhs_congr

-- 5. Aplicar la unicidad del resto euclídeo sobre numerales (euclidean_remainder_unique_numeral_Δ)

have hk12 := euclidean_remainder_unique_numeral_Δ Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ n1 n2 k_m k1 k2 hk1_lt hk2_lt h_num_eq

subst hk12

-- 6. Concluir $r_1 = r_2$ en Δ por transitividad y simetría

have h_r2_symm := eq_symm_Δ h_max h_con h_r2_eq

exact eq_trans_Δ h_max h_con h_r1_eq h_r2_symm

Lema `substForm_BetaForm_sh_unpack` y `beta_sh_functional_closed`

Se completa la infraestructura para `BetaForm_sh` con un lema de desempaquetado que simplifica su forma instanciada, y se demuestra la funcionalidad de la relación beta para términos cerrados arbitrarios.

Lema `substForm_BetaForm_sh_unpack`: Desempaquetado de `BetaForm_sh` instanciada

Establece la forma explícita que adopta `BetaForm_sh` cuando sus argumentos A, B, I se instancian con numerales y R con un término cerrado:

$$ \text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{I}, R) = \exists v_8\ (\overline{A} = v_8 \cdot M_0(I, B) + R \land R < M_0(I, B) \text{ (testigo 9)}) $$

La demostración es una simplificación directa de las sustituciones, utilizando que $\overline{A}$, $\overline{B}$, $\overline{I}$ son términos cerrados y por tanto inmunes a las sustituciones sobre las variables v_1, v_2, v_3. El resultado es la expansión completa de las definiciones de `BetaForm_sh`, `M0_term`, los cuantificadores y las fórmulas de igualdad y orden.

Lema `beta_sh_functional_closed`: Funcionalidad de `BetaForm_sh` para términos cerrados

Establece que la relación beta de Gödel es funcional en Δ: si dos términos cerrados R_1 y R_2 satisfacen `BetaForm_sh` con los mismos parámetros A, B, I, entonces son demostrablemente iguales:

$$ \Delta \vdash \text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{I}, R_1) \land \text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{I}, R_2) \implies \Delta \vdash R_1 = R_2 $$

Estructura de la demostración

La prueba es una aplicación magistral de todos los lemas desarrollados hasta ahora, organizada en siete etapas:

1. Desempaquetado: Se aplica `substForm_BetaForm_sh_unpack` a ambas hipótesis, obteniendo formas existenciales explícitas.

2. Extracción de cocientes: Por la propiedad de Henkin sobre la variable v_8, existen testigos cerrados Q_1 y Q_2 (los cocientes de la división) tales que:

- $\overline{A} = Q_1 \cdot M_0 + R_1 \in \Delta$ y $R_1 < M_0 \in \Delta$

- $\overline{A} = Q_2 \cdot M_0 + R_2 \in \Delta$ y $R_2 < M_0 \in \Delta$

3. Conversión de la fórmula de orden: Las condiciones $R_i < M_0$ están expresadas con variable testigo 9 (de `BetaForm_sh`). Se convierten a variable testigo 5 (requerida por los lemas posteriores) usando Henkin sobre v_9 y reintroducción del existencial sobre v_5.

4. Igualdad del módulo: Se establece que $M_0(I, B) = \overline{M_{\text{nat}}}$ en Δ (donde $M_{\text{nat}} = 1 + (I+1) \cdot B$), usando `M_term_eq_natToTerm`. Además, $M_{\text{nat}} \geq 1$.

5. Acotación y determinación de los cocientes: De $R_i < \overline{M_{\text{nat}}}$, por `lt_numeral_bounded` existen $k_i < M_{\text{nat}}$ con $R_i = \overline{k_i} \in \Delta$. Sustituyendo en la igualdad $\overline{A} = Q_i \cdot M_0 + R_i$ y usando $M_0 = \overline{M_{\text{nat}}}$, se obtiene $\overline{A} = Q_i \cdot \overline{M_{\text{nat}}} + \overline{k_i} \in \Delta$. Por el lema `Q'_bounded`, Q_i está acotado por $\overline{A+1}$, y por `lt_numeral_bounded` existe n_i tal que $Q_i = \overline{n_i} \in \Delta$.

6. Combinación de igualdades: De las dos igualdades para Q_1 y Q_2, por simetría y transitividad se obtiene $Q_1 \cdot M_0 + R_1 = Q_2 \cdot M_0 + R_2 \in \Delta$.

7. Aplicación de unicidad del resto euclídeo: Se invoca `euclid_remainder_unique_closed` con todos los parámetros ya determinados (M_0 es un numeral, Q_1, Q_2 son numerales, R_1, R_2 son numerales, ambos menores que M_0). La conclusión es $R_1 = R_2 \in \Delta$.

Significado

`beta_sh_functional_closed` es un resultado de una generalidad impresionante. Muestra que la funcionalidad de `BetaForm_sh` no solo vale para numerales (como en `beta_sh_representable`), sino para cualquier par de términos cerrados que satisfagan la relación beta con los mismos parámetros. La demostración integra prácticamente todas las herramientas desarrolladas en las fases anteriores:

- **Propiedad de Henkin** para extraer testigos existenciales (Q_1, Q_2, d_1, d_2).
- **`lt_numeral_bounded`** para reducir términos cerrados a numerales cuando satisfacen condiciones de orden.
- **`Q'_bounded`** para acotar cocientes y reducirlos también a numerales.
- **Manipulaciones algebraicas** con `eq_mul_Δ`, `eq_add_Δ`, `eq_trans_Δ`, `eq_symm_Δ` para reescribir igualdades.
- **`euclid_remainder_unique_closed`** como resultado final de unicidad.

- **`M_term_eq_natToTerm`** para conectar la representación del módulo como término compuesto con su representación como numeral.

Esta integración es un testimonio de la madurez alcanzada por la infraestructura deductiva desarrollada para Δ. El lema será utilizado de manera crucial en la demostración de funcionalidad de `NumForm`, donde las condiciones de `BetaForm_sh` aparecen repetidamente para diferentes valores de los parámetros.

/-- Helper para desempaquetar la sustitución sobre BetaForm_sh -/

lemma substForm_BetaForm_sh_unpack {s : Nat} (A B I : Nat) (R : TermClosed s) :

  substForm 3 R.val (substForm 2 (natToTerm (s := s) I) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) =

  exists_form 8 (and_form

   (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) (M0_term I B)) R.val))

   (lt_form R.val (M0_term I B) 9)) := by

 dsimp [BetaForm_sh, M0_term, exists_form, ArithForm.exists_form, and_form, lt_form, substForm, substTerm]

 rw [substTerm_fresh (natToTerm A) 1 (natToTerm B) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 1),

   substTerm_fresh (natToTerm A) 2 (natToTerm I) (NotOccur_of_isClosed _ (isClosed_natToTerm A) 2),

   substTerm_fresh (natToTerm A) 3 R.val (NotOccur_of_isClosed _ (isClosed_natToTerm A) 3),

   substTerm_fresh (natToTerm B) 2 (natToTerm I) (NotOccur_of_isClosed _ (isClosed_natToTerm B) 2),

```
    substTerm_fresh (natToTerm B) 3 R.val (NotOccur_of_isClosed _ (isClosed_natToTerm B) 3),
    substTerm_fresh (natToTerm I) 3 R.val (NotOccur_of_isClosed _ (isClosed_natToTerm I) 3)]


/--
   Paso 1.3: beta_sh_functional_closed.
   Demuestra que la relación beta de Gödel BetaForm_sh es funcional en Δ
   para términos cerrados R1 y R2.
-/
lemma beta_sh_functional_closed {s : Nat} (Δ : Set (ArithForm s))
   (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)
   (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)
   (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)
   (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)
   (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)
   (A B I : Nat) (R1 R2 : TermClosed s)
   (h1 : substForm 3 R1.val (substForm 2 (natToTerm (s := s) I)
      (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) ∈ Δ)
   (h2 : substForm 3 R2.val (substForm 2 (natToTerm (s := s) I)
      (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) ∈ Δ) :
   ArithForm.eq R1.val R2.val ∈ Δ := by
  let M_c : TermClosed s := ⟨M0_term I B, isClosed_M0 I B⟩
  let M_nat := 1 + (I + 1) * B

  -- 1. Desempaquetar h1 y h2
  rw [substForm_BetaForm_sh_unpack A B I R1] at h1
  rw [substForm_BetaForm_sh_unpack A B I R2] at h2
```

```
  -- 2. Extraer cocientes Q1 y Q2 por Henkin
  obtain ⟨Q1, hQ1⟩ := h_henkin 8 _ h1
  have h_nf_lt1 : NotFreeForm 8 (lt_form R1.val (M0_term (s := s) I B) 9) := by
    dsimp [lt_form, exists_form, ArithForm.exists_form, NotFreeForm]
    exact ⟨NotOccur_M0 I B 8,
      NotOccur.add _ _ (NotOccur_of_isClosed _ R1.property 8) (NotOccur.succ _ (NotOccur.var 9 (by decide)))⟩
  have hQ1_simp : substForm 8 Q1.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) (M0_term (s := s) I B)) R1.val)) (lt_form R1.val (M0_term (s := s) I B) 9)) =
      and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul Q1.val (M0_term (s := s) I B)) R1.val)) (lt_form R1.val (M0_term (s := s) I B) 9) := by
    unfold and_form
    rw [substForm_neg, substForm_impl, substForm_neg]
    rw [substForm_fresh_of_NotFreeForm _ 8 Q1.val h_nf_lt1]
    rw [substForm_eq]
    dsimp [substTerm]
    rw [substTerm_fresh (natToTerm A) 8 Q1.val (NotOccur_of_isClosed _ (isClosed_natToTerm A) 8),
      substTerm_fresh (M0_term I B) 8 Q1.val (NotOccur_M0 I B 8),
      substTerm_fresh R1.val 8 Q1.val (NotOccur_of_isClosed _ R1.property 8)]
  rw [hQ1_simp] at hQ1
  have h_eq1 := and_elim_left_Δ h_max h_con _ _ hQ1
  have h_lt1_9 := and_elim_right_Δ h_max h_con _ _ hQ1

  obtain ⟨Q2, hQ2⟩ := h_henkin 8 _ h2
  have h_nf_lt2 : NotFreeForm 8 (lt_form R2.val (M0_term (s := s) I B) 9) := by
    dsimp [lt_form, exists_form, ArithForm.exists_form, NotFreeForm]
```

exact ⟨NotOccur_M0 I B 8,

NotOccur.add _ _ (NotOccur_of_isClosed _ R2.property 8) (NotOccur.succ _ (NotOccur.var 9 (by decide)))⟩

have hQ2_simp : substForm 8 Q2.val (and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul (ArithTerm.var 8) (M0_term (s := s) I B)) R2.val)) (lt_form R2.val (M0_term (s := s) I B) 9)) =

and_form (ArithForm.eq (natToTerm (s := s) A) (ArithTerm.add (ArithTerm.mul Q2.val (M0_term (s := s) I B)) R2.val)) (lt_form R2.val (M0_term (s := s) I B) 9) := by

unfold and_form

rw [substForm_neg, substForm_impl, substForm_neg]

rw [substForm_fresh_of_NotFreeForm _ 8 Q2.val h_nf_lt2]

rw [substForm_eq]

dsimp [substTerm]

rw [substTerm_fresh (natToTerm A) 8 Q2.val (NotOccur_of_isClosed _ (isClosed_natToTerm A) 8),

substTerm_fresh (M0_term I B) 8 Q2.val (NotOccur_M0 I B 8),

substTerm_fresh R2.val 8 Q2.val (NotOccur_of_isClosed _ R2.property 8)]

rw [hQ2_simp] at hQ2

have h_eq2 := and_elim_left_Δ h_max h_con _ _ hQ2

have h_lt2_9 := and_elim_right_Δ h_max h_con _ _ hQ2

-- 3. Convertir lt_form ... 9 a lt_form ... 5

obtain ⟨d1, hd1⟩ := h_henkin 9 (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R1.val (ArithTerm.succ (ArithTerm.var 9)))) h_lt1_9

have hd1_simp : substForm 9 d1.val (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R1.val (ArithTerm.succ (ArithTerm.var 9)))) =

ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R1.val (ArithTerm.succ d1.val)) := by

dsimp [substForm, substTerm]

rw [substTerm_fresh (M0_term I B) 9 d1.val (NotOccur_M0 I B 9),

substTerm_fresh R1.val 9 d1.val (NotOccur_of_isClosed _ R1.property 9)]

```
 rw [hd1_simp] at hd1

 have h_subst1 : substForm 5 d1.val (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R1.val (ArithTerm.succ (ArithTerm.var 5)))) =

          ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R1.val (ArithTerm.succ d1.val)) := by

   dsimp [substForm, substTerm]

   rw [substTerm_fresh (M0_term I B) 5 d1.val (NotOccur_M0 I B 5),

      substTerm_fresh R1.val 5 d1.val (NotOccur_of_isClosed _ R1.property 5)]

 have h_subst1_in : substForm 5 d1.val (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R1.val (ArithTerm.succ (ArithTerm.var 5)))) ∈ Δ := by

   rwa [h_subst1]

 have hfree1 : IsFreeFor 5 d1.val (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R1.val (ArithTerm.succ (ArithTerm.var 5)))) := IsFreeFor_closed _ 5 d1

 have hex1 := exists_intro Δ 5 _ d1.val hfree1 (SPA_Deduction_Internal.hyp Δ _ h_subst1_in)

 have h_lt1_5 : lt_form R1.val (M0_term (s := s) I B) 5 ∈ Δ := maximal_consistent_closed_under_deduction Δ h_max h_con _ hex1


 obtain ⟨d2, hd2⟩ := h_henkin 9 (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R2.val (ArithTerm.succ (ArithTerm.var 9)))) h_lt2_9

 have hd2_simp : substForm 9 d2.val (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R2.val (ArithTerm.succ (ArithTerm.var 9)))) =

          ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R2.val (ArithTerm.succ d2.val)) := by

   dsimp [substForm, substTerm]

   rw [substTerm_fresh (M0_term I B) 9 d2.val (NotOccur_M0 I B 9),

      substTerm_fresh R2.val 9 d2.val (NotOccur_of_isClosed _ R2.property 9)]

 rw [hd2_simp] at hd2

 have h_subst2 : substForm 5 d2.val (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R2.val (ArithTerm.succ (ArithTerm.var 5)))) =

          ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R2.val (ArithTerm.succ d2.val)) := by
```

```
  dsimp [substForm, substTerm]

  rw [substTerm_fresh (M0_term I B) 5 d2.val (NotOccur_M0 I B 5),

    substTerm_fresh R2.val 5 d2.val (NotOccur_of_isClosed _ R2.property 5)]

 have h_subst2_in : substForm 5 d2.val (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R2.val (ArithTerm.succ (ArithTerm.var 5)))) ∈ Δ := by

  rwa [h_subst2]

 have hfree2 : IsFreeFor 5 d2.val (ArithForm.eq (M0_term (s := s) I B) (ArithTerm.add R2.val (ArithTerm.succ (ArithTerm.var 5)))) := IsFreeFor_closed _ 5 d2

 have hex2 := exists_intro Δ 5 _ d2.val hfree2 (SPA_Deduction_Internal.hyp Δ _ h_subst2_in)

 have h_lt2_5 : lt_form R2.val (M0_term (s := s) I B) 5 ∈ Δ := maximal_consistent_closed_under_deduction Δ h_max h_con _ hex2


 -- 4. Probar que M0_term I B = natToTerm M_nat en Δ

 have h_M_eq := M_term_eq_natToTerm Δ h_max h_con h_add_zero h_add_succ h_mul_zero h_mul_succ I B

 have h_M_symm := eq_symm_Δ h_max h_con h_M_eq

 have h_M_pos : ∃ k, k ≥ 1 ∧ ArithForm.eq M_c.val (natToTerm (s := s) k) ∈ Δ :=

  ⟨M_nat, by omega, h_M_symm⟩


 -- 5. Probar que Q1 y Q2 son acotados y por lo tanto iguales a numerales

 have h_lt1_nat := lt_form_M_term_to_natToTerm Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ I B R1 h_lt1_5

 obtain ⟨k1, hk1_lt, h_R1_k1⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ M_nat R1 h_lt1_nat


 have h_congr_M1 := eq_mul_Δ h_max h_con (eq_refl_in_Δ Δ h_max h_con Q1.val) h_M_symm

 have h_congr_add1 := eq_add_Δ h_max h_con h_congr_M1 h_R1_k1

 have h_eq1_num := eq_trans_Δ h_max h_con h_eq1 h_congr_add1
```

have h_Q1_bound := Q'_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ A M_nat k1 Q1 h_eq1_num (by omega)

obtain ⟨n1, _, h_Q1_n1⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ (A + 1) Q1 h_Q1_bound

have hq1 : ∃ n1, ArithForm.eq Q1.val (natToTerm (s := s) n1) ∈ Δ := ⟨n1, h_Q1_n1⟩

have h_lt2_nat := lt_form_M_term_to_natToTerm Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ I B R2 h_lt2_5

obtain ⟨k2, hk2_lt, h_R2_k2⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ M_nat R2 h_lt2_nat

have h_congr_M2 := eq_mul_Δ h_max h_con (eq_refl_in_Δ Δ h_max h_con Q2.val) h_M_symm

have h_congr_add2 := eq_add_Δ h_max h_con h_congr_M2 h_R2_k2

have h_eq2_num := eq_trans_Δ h_max h_con h_eq2 h_congr_add2

have h_Q2_bound := Q'_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ A M_nat k2 Q2 h_eq2_num (by omega)

obtain ⟨n2, _, h_Q2_n2⟩ := lt_numeral_bounded Δ h_max h_con h_henkin h_add_zero h_add_succ (A + 1) Q2 h_Q2_bound

have hq2 : ∃ n2, ArithForm.eq Q2.val (natToTerm (s := s) n2) ∈ Δ := ⟨n2, h_Q2_n2⟩

-- 6. Combinar h_eq1 y h_eq2

have h_eq1_symm := eq_symm_Δ h_max h_con h_eq1

have h_eq_comb := eq_trans_Δ h_max h_con h_eq1_symm h_eq2

-- 7. Aplicar euclid_remainder_unique_closed (Paso 1.2)

exact euclid_remainder_unique_closed Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ Q1 Q2 R1 R2 M_c h_M_pos hq1 hq2 h_eq_comb h_lt1_5 h_lt2_5

Definiciones auxiliares y lema de frescura para la demostración de funcionalidad de `NumForm`

Se definen tres fórmulas que encapsulan partes específicas de `NumForm` para facilitar la demostración de funcionalidad, y se establece un lema de frescura necesario para la manipulación de sustituciones.

`C1_formula`: Caso base de la secuencia

Define la fórmula que expresa que el primer elemento de la secuencia codificada por (A, B) es el código de `zero`:

$$ \text{C1_formula}(A, B) := \text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{0}, \overline{\text{zeroCode}}) $$

`C3_formula`: Condición inductiva de la secuencia

Define la fórmula que expresa que la secuencia codificada por (A, B) respeta la relación de sucesor para todo $k < n$:

$$ \text{C3_formula}(A, B, n) := \forall k\ (k < \overline{n} \to \exists v, v'\ (\text{BetaForm_sh}(\overline{A}, \overline{B}, k, v) \land \text{BetaForm_sh}(\overline{A}, \overline{B}, S(k), v') \land \text{IsSuccCode}(v, v'))) $$

`CK_formula`: Valor de la secuencia en la posición k

Define la fórmula que expresa que el k-ésimo elemento de la secuencia es $\text{seq_code}(k)$:

$$ \text{CK_formula}(A, B, k) := \text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k}, \overline{\text{seq_code}(k)}) $$

Lema `NotFreeForm_C4_V`: Frescura para `BetaForm_sh` con término cerrado genérico

Generaliza los lemas de frescura previos para `BetaForm_sh` instanciada, permitiendo que el cuarto argumento sea un término cerrado arbitrario V en lugar de un numeral:

$$ \forall A, B, k \in \mathbb{N},\ \forall V \in \text{TermClosed } s,\ \forall v \notin \{0,1,2,3,8,9\}, $$

$$ \text{NotFreeForm } v\ (\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k}, V)) $$

La demostración sigue el mismo patrón que los lemas `NotFreeForm_C4_closed` y `NotFreeForm_C5_closed`, pero reemplazando el numeral $\overline{v_k}$ por el término cerrado V. La propiedad fundamental es que V, al ser cerrado, no contiene ninguna variable libre, por lo que `NotOccur_of_isClosed` garantiza que ninguna variable v ocurre en V.

Significado para la funcionalidad de `NumForm`

Estas definiciones y lemas son los preparativos finales para la demostración de que `NumForm` satisface la condición de funcionalidad. La estrategia de la prueba será:

1. Asumir $\text{NumForm}(\overline{n}, Z) \in \Delta$ para algún término cerrado Z.

2. Por `NumForm_subst_closed`, esto equivale a la existencia de parámetros A, B tales que:

- `C1_formula(A, B)` (el caso base)

- $\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{n}, Z)$ (el elemento final)

- `C3_formula(A, B, n)` (la condición inductiva)

3. Por `beta_sh_functional_closed`, aplicado a la igualdad del elemento final y al hecho conocido de que $\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{n}, \overline{\text{seq_code}(n)}) \in \Delta$ (por `num_representable_graph`), se obtendrá $Z = \overline{\text{seq_code}(n)}$.

El lema `NotFreeForm_C4_V` garantiza que durante la manipulación de las subfórmulas dentro de `C3_formula`, las variables ligadas de `NumForm` no interfieren con las variables libres de `BetaForm_sh` cuando se sustituyen términos cerrados. Esto permite aplicar con seguridad los lemas de representabilidad y funcionalidad a cada instancia de `BetaForm_sh` dentro de la fórmula cuantificada universalmente.

/-- Fórmula que representa el caso base c1 -/

def C1_formula {s : Nat} (A B : Nat) : ArithForm s :=

substForm 3 (natToTerm (s := s) zeroCode) (substForm 2 (natToTerm (s := s) 0)

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh)))

/-- Fórmula cuantificada universalmente c3 -/

```
def C3_formula {s : Nat} (A B n : Nat) : ArithForm s :=
 ArithForm.forall_form 4 (ArithForm.impl (lt_form (ArithTerm.var (s := s) 4) (natToTerm (s := s) n) 11)
   (exists_form 5 (exists_form 6 (and_form
    (substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (ArithTerm.var (s := s) 4)
     (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))
    (and_form
     (substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))
      (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))
     IsSuccCode)))))

/-- Fórmula objetivo para el k-ésimo paso de la secuencia -/
def CK_formula {s : Nat} (A B k : Nat) : ArithForm s :=
 substForm 3 (natToTerm (s := s) (seq_code k)) (substForm 2 (natToTerm (s := s) k)
  (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh)))

/-- Lema de no ocurrencia libre de v en la subfórmula C4 para un término V cerrado genérico -/
lemma NotFreeForm_C4_V {s : Nat} (A B k : Nat) (V : TermClosed s) (v : Nat)
   (hv0 : v ≠ 0) (hv1 : v ≠ 1) (hv2 : v ≠ 2) (hv3 : v ≠ 3) (hv8 : v ≠ 8) (hv9 : v ≠ 9) :
   NotFreeForm v (substForm 3 V.val
    (substForm 2 (natToTerm (s := s) k)
     (substForm 1 (natToTerm (s := s) B)
      (substForm 0 (natToTerm (s := s) A) BetaForm_sh)))) := by
 apply NotFreeForm_of_NotOccurForm
 apply NotOccurForm_subst_other
```

· apply NotOccurForm_subst_other

· apply NotOccurForm_subst_other

· apply NotOccurForm_subst_other

· dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, NotOccurForm]

refine ⟨hv8, ⟨⟨NotOccur.var 0 hv0, ?_⟩, ⟨hv9, ⟨?_, ?_⟩⟩⟩⟩

· apply NotOccur.add

· apply NotOccur.mul

· exact NotOccur.var 8 hv8

· apply NotOccur.succ; apply NotOccur.mul

· apply NotOccur.succ; exact NotOccur.var 2 hv2

· exact NotOccur.var 1 hv1

· exact NotOccur.var 3 hv3

· apply NotOccur.succ; apply NotOccur.mul

· apply NotOccur.succ; exact NotOccur.var 2 hv2

· exact NotOccur.var 1 hv1

· apply NotOccur.add

· exact NotOccur.var 3 hv3

· apply NotOccur.succ; exact NotOccur.var 9 hv9

· exact NotOccur_of_isClosed _ (isClosed_natToTerm A) v

· exact NotOccur_of_isClosed _ (isClosed_natToTerm B) v

· exact NotOccur_of_isClosed _ (isClosed_natToTerm k) v

· exact NotOccur_of_isClosed _ V.property v

Lemas de cadena de sustitución directa para `NumForm` / `BetaForm_sh`

Se demuestran cuatro lemas que simplifican las sustituciones encadenadas que aparecen al instanciar la variable ligada v_4 (la variable de inducción k) por un numeral concreto $\overline{k'}$ dentro de las subfórmulas de `NumForm`. Estos lemas son esenciales para la demostración de funcionalidad, donde será necesario evaluar la condición inductiva para valores específicos de k.

Lema `substForm_lt_chain`: Sustitución en la fórmula de orden

Establece que al sustituir v_4 por $\overline{k'}$ en la condición $v_4 < \overline{n}$ (con variable testigo 11), se obtiene $\overline{k'} < \overline{n}$:

$$ (v_4 < \overline{n})[v_4 := \overline{k'}] = \overline{k'} < \overline{n} $$

La demostración es directa: el término $\overline{n}$ es cerrado, por lo que la sustitución no lo afecta, y la variable v_4 se reemplaza por $\overline{k'}$.

Lema `substForm_Beta_chain1`: Sustitución en `BetaForm_sh` para el valor v

Establece que al sustituir v_4 por $\overline{k'}$ en la fórmula $\text{BetaForm_sh}(\overline{A}, \overline{B}, v_4, v_5)$, se obtiene $\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k'}, v_5)$:

$$ \text{BetaForm_sh}(\overline{A}, \overline{B}, v_4, v_5)[v_4 := \overline{k'}] = \text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k'}, v_5) $$

La demostración expande las definiciones y aplica repetidamente `substTerm_fresh` para eliminar las sustituciones sobre los términos cerrados $\overline{A}$, $\overline{B}$, $\overline{k'}$.

Lema `substForm_Beta_chain2`: Sustitución en `BetaForm_sh` para el valor v'

Análogo al anterior, pero para la fórmula con $S(v_4)$ en la posición del índice:

$$ \text{BetaForm_sh}(\overline{A}, \overline{B}, S(v_4), v_6)[v_4 := \overline{k'}] = \text{BetaForm_sh}(\overline{A}, \overline{B}, S(\overline{k'}), v_6) $$

Lema `substForm_phi_body_4`: Sustitución completa en el cuerpo de la condición inductiva

Combina los tres lemas anteriores para dar la forma explícita de la sustitución de v_4 por $\overline{k'}$ en la fórmula φ_{body} (la implicación dentro de `C3_formula`):

$$ \varphi_{\text{body}}[v_4 := \overline{k'}] = (\overline{k'} < \overline{n}) \to \exists v_5, v_6\ (\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k'}, v_5) \land \text{BetaForm_sh}(\overline{A}, \overline{B}, S(\overline{k'}), v_6) \land \text{IsSuccCode}(v_5, v_6)) $$

La demostración es una reescritura directa utilizando los tres lemas anteriores y el hecho de que `IsSuccCode` no contiene a v_4, por lo que la sustitución no la afecta.

Significado para la funcionalidad de `NumForm`

Estos lemas de cadena de sustitución son herramientas técnicas que permiten "evaluar" la condición inductiva de `NumForm` para valores concretos de k. En la demostración de funcionalidad, después de extraer los parámetros A, B mediante Henkin, se tendrá que $\forall k\ \varphi_{\text{body}}(k) \in \Delta$. Para utilizar esta información, será necesario instanciar el cuantificador universal con numerales específicos $\overline{k'}$ (para $k' = 0, 1, \ldots, n-1$) y obtener las condiciones correspondientes.

La eliminación del cuantificador universal (mediante `forall_elim`) produce $\varphi_{\text{body}}[v_4 := \overline{k'}] \in \Delta$. Los lemas de cadena de sustitución permiten reescribir esta fórmula en una forma donde todas las referencias a v_4 han sido reemplazadas por $\overline{k'}$, haciendo explícitas las condiciones sobre `BetaForm_sh` e `IsSuccCode` para el k'-ésimo paso. Esta forma explícita es necesaria para aplicar los lemas de representabilidad y funcionalidad (`beta_sh_representable`, `beta_sh_functional_closed`, `IsSuccCode_representable`) que esperan argumentos numerales concretos en lugar de variables.

```
--
================================================================

-- LEMAS DE CADENA DE SUSTITUCIÓN DIRECTA PARA NumForm / BetaForm_sh

--
================================================================


lemma substForm_lt_chain {s : Nat} (n k' : Nat) :

  substForm 4 (natToTerm (s := s) k') (lt_form (ArithTerm.var (s := s) 4) (natToTerm (s := s) n) 11) =

  lt_form (natToTerm (s := s) k') (natToTerm (s := s) n) 11 := by

 dsimp [lt_form, exists_form, ArithForm.exists_form, substForm, substTerm]

 repeat rw [substTerm_fresh (natToTerm n) _ _ (NotOccur_of_isClosed _ (isClosed_natToTerm n) _)]


lemma substForm_Beta_chain1 {s : Nat} (A B k' : Nat) :

  substForm 4 (natToTerm (s := s) k')

    (substForm 3 (ArithTerm.var (s := s) 5)

      (substForm 2 (ArithTerm.var (s := s) 4)

        (substForm 1 (natToTerm (s := s) B)

          (substForm 0 (natToTerm (s := s) A) BetaForm_sh)))) =

  substForm 3 (ArithTerm.var (s := s) 5)

    (substForm 2 (natToTerm (s := s) k')

      (substForm 1 (natToTerm (s := s) B)

        (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) := by

 dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, substForm, substTerm]

 repeat rw [substTerm_fresh (natToTerm A) _ _ (NotOccur_of_isClosed _ (isClosed_natToTerm A) _)]

 repeat rw [substTerm_fresh (natToTerm B) _ _ (NotOccur_of_isClosed _ (isClosed_natToTerm B) _)]
```

```
 rw [substTerm_fresh (natToTerm k') 3 (ArithTerm.var 5) (NotOccur_of_isClosed _ (isClosed_natToTerm k') 3)]


lemma substForm_Beta_chain2 {s : Nat} (A B k' : Nat) :
  substForm 4 (natToTerm (s := s) k')
   (substForm 3 (ArithTerm.var (s := s) 6)
    (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))
     (substForm 1 (natToTerm (s := s) B)
      (substForm 0 (natToTerm (s := s) A) BetaForm_sh)))) =
  substForm 3 (ArithTerm.var (s := s) 6)
   (substForm 2 (ArithTerm.succ (natToTerm (s := s) k'))
    (substForm 1 (natToTerm (s := s) B)
     (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) := by
 dsimp [BetaForm_sh, exists_form, and_form, lt_form, ArithForm.exists_form, substForm, substTerm]
 repeat rw [substTerm_fresh (natToTerm A) _ _ (NotOccur_of_isClosed _ (isClosed_natToTerm A) _)]
 repeat rw [substTerm_fresh (natToTerm B) _ _ (NotOccur_of_isClosed _ (isClosed_natToTerm B) _)]
 rw [substTerm_fresh (natToTerm k') 3 (ArithTerm.var 6) (NotOccur_of_isClosed _ (isClosed_natToTerm k') 3)]


/-- Helper unificado de la sustitución de la variable 4 por natToTerm k' en φ_body -/
lemma substForm_phi_body_4 {s : Nat} (A B n k' : Nat) :
  let φ_body : ArithForm s := ArithForm.impl (lt_form (ArithTerm.var (s := s) 4) (natToTerm (s := s) n) 11)
   (exists_form 5 (exists_form 6 (and_form
    (substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (ArithTerm.var (s := s) 4)
     (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))
    (and_form
```

```
      (substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ
(ArithTerm.var (s := s) 4))

       (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A)
BetaForm_sh))))

      IsSuccCode))))

  substForm 4 (natToTerm (s := s) k') φ_body =

   ArithForm.impl (lt_form (natToTerm (s := s) k') (natToTerm (s := s) n) 11)

    (exists_form 5 (exists_form 6 (and_form

     (substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (natToTerm (s := s) k')

      (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A)
BetaForm_sh))))

      (and_form

       (substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ
(natToTerm (s := s) k'))

        (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A)
BetaForm_sh))))

      IsSuccCode)))) := by

 intro φ_body

 dsimp [φ_body, exists_form, ArithForm.exists_form, and_form, substForm,
substTerm]

 rw [substForm_lt_chain n k']

 rw [substForm_Beta_chain1 A B k']

 rw [substForm_Beta_chain2 A B k']

 rw [substForm_fresh_of_NotFreeForm IsSuccCode 4 (natToTerm k')
(NotFreeForm_IsSuccCode 4 (by decide) (by decide))]
```

Lema `num_representable_induction`: Inducción sobre la secuencia de códigos

Se demuestra el lema central que establece que si los parámetros A, B satisfacen las condiciones `C1` (caso base) y `C3` (condición inductiva) de `NumForm` en Δ, entonces para todo $k \leq n$, la fórmula `CK_formula(A, B, k)` —que expresa $\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k}, \overline{\text{seq_code}(k)})$— pertenece a Δ:

$$ \forall A, B, n \in \mathbb{N},\ \text{C1_formula}(A, B) \in \Delta \land \text{C3_formula}(A, B, n) \in \Delta \implies \forall k \leq n,\ \text{CK_formula}(A, B, k) \in \Delta $$

Estructura de la demostración

La prueba procede por inducción en el metalenguaje sobre k:

Caso base ($k = 0$): `CK_formula(A, B, 0)` es exactamente `C1_formula(A, B)`, que está en Δ por hipótesis.

Paso inductivo ($k = k' + 1$): Se asume que `CK_formula(A, B, k')` está en Δ (hipótesis de inducción) y se debe demostrar que `CK_formula(A, B, k' + 1)` también lo está.

1. **Instanciación del cuantificador universal en `C3`:** La hipótesis `C3` es $\forall v_4\ \varphi_{\text{body}}(v_4)$. Se instancia v_4 con el numeral $\overline{k'}$ (verificando que es libre para v_4 en φ_{body}), obteniendo $\varphi_{\text{body}}[v_4 := \overline{k'}] \in \Delta$.

2. **Simplificación mediante `substForm_phi_body_4`:** Se aplica el lema de cadena de sustitución para reescribir la fórmula instanciada como:

$$ (\overline{k'} < \overline{n}) \to \exists v_5, v_6\ (\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k'}, v_5) \land \text{BetaForm_sh}(\overline{A}, \overline{B}, S(\overline{k'}), v_6) \land \text{IsSuccCode}(v_5, v_6)) $$

3. **Verificación del antecedente:** Dado que $k' < n$ (pues $k = k'+1 \leq n$), se aplica `lt_form_intro_w` para obtener $\overline{k'} < \overline{n} \in \Delta$.

4. **Modus ponens:** Por `maximal_consistent_imp_closed`, el consecuente (la fórmula existencial) pertenece a Δ.

5. **Extracción de testigos:** Aplicando la propiedad de Henkin dos veces (sobre v_5 y luego sobre v_6), se obtienen términos cerrados V y V' tales que:

- $\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k'}, V) \in \Delta$
- $\text{BetaForm_sh}(\overline{A}, \overline{B}, S(\overline{k'}), V') \in \Delta$
- $\text{IsSuccCode}(V, V') \in \Delta$

Las simplificaciones de las sustituciones utilizan los lemas de frescura `NotFreeForm_C4_V` y `NotFreeForm_C5_var6` para conmutar correctamente las sustituciones.

6. **Determinación de V:** Por la hipótesis de inducción, $\text{BetaForm_sh}(\overline{A}, \overline{B}, \overline{k'}, \overline{\text{seq_code}(k')}) \in \Delta$. Aplicando `beta_sh_functional_closed` a esta fórmula y a la primera condición del paso anterior, se obtiene $V = \overline{\text{seq_code}(k')} \in \Delta$.

7. **Determinación de V' vía `IsSuccCode`:** Se sustituye V por $\overline{\text{seq_code}(k')}$ en la condición `IsSuccCode(V, V')` usando la regla `eq_subst`. Luego, por la funcionalidad de `IsSuccCode_representable` (específicamente, la segunda componente del par devuelto), se obtiene $V' = \overline{\text{seq_code}(k' + 1)} \in \Delta$, ya que $\text{seq_code}(k' + 1) = \text{pair}(4, \text{seq_code}(k')) + 1$.

8. **Sustitución final:** Se reemplaza V' por $\overline{\text{seq_code}(k' + 1)}$ en la condición $\text{BetaForm_sh}(\overline{A}, \overline{B}, S(\overline{k'}), V')$ usando `eq_subst`, obteniendo exactamente `CK_formula(A, B, k' + 1) \in \Delta$.

Significado

`num_representable_induction` es el resultado que conecta la condición inductiva `C3` (que afirma la existencia de algún testigo para cada paso) con la conclusión fuerte de que los testigos son exactamente los códigos de los numerales $\text{seq_code}(k)$. La demostración es una obra maestra de integración: combina la inducción en el metalenguaje, la eliminación de cuantificadores, la propiedad de Henkin, la funcionalidad de `BetaForm_sh` (`beta_sh_functional_closed`), la representabilidad y funcionalidad de `IsSuccCode`, y las reglas de sustitución de iguales (`eq_subst`).

Este lema muestra que la condición `C3` es suficiente para forzar que toda la secuencia (A, B) coincida exactamente con la secuencia de códigos de numerales. Es el análogo en el sistema formal del hecho de que si una secuencia comienza con el código de `zero` y cada paso respeta la relación `IsSuccCode`, entonces la secuencia es precisamente la de los códigos de los numerales. Esta unicidad forzosa es la esencia de la funcionalidad de `NumForm`.

/--

Paso 1.4: num_representable_induction.

Demuestra por inducción en el metalenguaje sobre k ≤ n que la secuencia

β(A, B, k) equivale a natToTerm (seq_code k) en Δ.

-/

```
lemma num_representable_induction {s : Nat} (Δ : Set (ArithForm s))
  (h_max : Maximal Δ) (h_con : Consistent Δ) (h_henkin : HenkinSet Δ)
  (h_add_zero : ∀ x, add_zero_axiom x ∈ Δ)
  (h_add_succ : ∀ x y, add_succ_axiom x y ∈ Δ)
  (h_mul_zero : ∀ x, mul_zero_axiom x ∈ Δ)
  (h_mul_succ : ∀ x y, mul_succ_axiom x y ∈ Δ)
  (A B : Nat) (n : Nat)
  (hC1 : C1_formula A B ∈ Δ)
  (hC3 : C3_formula A B n ∈ Δ) :
  ∀ k, k ≤ n → CK_formula A B k ∈ Δ := by
 intro k
 induction k with
 | zero =>
  intro _
  exact hC1
 | succ k' ih =>
  intro hk1
  have hk' : k' ≤ n := by omega
  have ih_k' := ih hk'
  have hk'_lt : k' < n := by omega


  -- 1. Instanciar hC3 con natToTerm k'
  let φ_body : ArithForm s := ArithForm.impl (lt_form (ArithTerm.var (s := s) 4) (natToTerm (s := s) n) 11)
    (exists_form 5 (exists_form 6 (and_form
      (substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (ArithTerm.var (s := s) 4)
       (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))
      (and_form
```

(substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (ArithTerm.var (s := s) 4))

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

IsSuccCode))))

have h_C3_eq : C3_formula A B n = ArithForm.forall_form 4 φ_body := rfl

have hC3' : ArithForm.forall_form 4 φ_body ∈ Δ := by rwa [← h_C3_eq]

have h_free_body : IsFreeFor 4 (natToTerm (s := s) k') φ_body := IsFreeFor_closed _ 4 ⟨natToTerm k', isClosed_natToTerm k'⟩

have h_elim_body := SPA_Deduction_Internal.forall_elim Δ 4 φ_body (natToTerm k') (SPA_Deduction_Internal.hyp Δ _ hC3') h_free_body

have h_elim_in := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_elim_body

-- 2. Deducir lt_form (natToTerm k') (natToTerm n) 11 ∈ Δ

have h_lt_k' := lt_form_intro_w Δ h_max h_con h_add_zero h_add_succ k' n hk'_lt 11

-- 3. Simplificar la sustitución sobre φ_body mediante substForm_phi_body_4

have h_subst_phi := substForm_phi_body_4 (s := s) A B n k'

rw [h_subst_phi] at h_elim_in

have h_ex56 := maximal_consistent_imp_closed Δ h_max h_con _ _ h_lt_k' h_elim_in

-- 4. Extraer testigos Henkin V y V'

let body5 := exists_form 6 (and_form

(substForm 3 (ArithTerm.var (s := s) 5) (substForm 2 (natToTerm (s := s) k')

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(and_form

(substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (natToTerm (s := s) k'))

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

IsSuccCode))

obtain ⟨V, hV⟩ := h_henkin 5 body5 h_ex56

have h_occ5_AB : NotOccurForm 5 (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh)) :=

NotOccurForm_Beta_subst_AB A B 5 (by decide) (by decide) (by decide) (by decide) (by decide) (by decide)

have h_occ5_C4 : NotOccurForm 5 (substForm 2 (natToTerm (s := s) k') (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))) :=

NotOccurForm_subst_other _ 2 5 (natToTerm k') h_occ5_AB (NotOccur_of_isClosed _ (isClosed_natToTerm k') 5)

have hV_simp : substForm 5 V.val body5 =

exists_form 6 (and_form

(substForm 3 V.val (substForm 2 (natToTerm (s := s) k')

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(and_form

(substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ (natToTerm (s := s) k'))

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(substForm 5 V.val IsSuccCode))) := by

dsimp [body5, exists_form, ArithForm.exists_form, and_form, substForm, substTerm]

rw [substForm_substForm_term_comm _ 3 5 V.val h_occ5_C4]

```
    rw [substForm_fresh_of_NotFreeForm _ 5 V.val (NotFreeForm_C5_var6 A B k' 5 (by
decide) (by decide) (by decide) (by decide) (by decide) (by decide) (by decide))]


   rw [hV_simp] at hV

   obtain ⟨V', hV'⟩ := h_henkin 6 _ hV


   have h_occ6_AB : NotOccurForm 6 (substForm 1 (natToTerm (s := s) B) (substForm
0 (natToTerm (s := s) A) BetaForm_sh)) :=

    NotOccurForm_Beta_subst_AB A B 6 (by decide) (by decide) (by decide) (by
decide) (by decide) (by decide)

   have h_occ6_C5 : NotOccurForm 6 (substForm 2 (ArithTerm.succ (natToTerm (s :=
s) k')) (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A)
BetaForm_sh))) :=

    NotOccurForm_subst_other _ 2 6 (ArithTerm.succ (natToTerm k')) h_occ6_AB
(NotOccur.succ _ (NotOccur_of_isClosed _ (isClosed_natToTerm k') 6))


   have hV'_simp : substForm 6 V'.val (and_form

     (substForm 3 V.val (substForm 2 (natToTerm (s := s) k')

      (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A)
BetaForm_sh))))

     (and_form

      (substForm 3 (ArithTerm.var (s := s) 6) (substForm 2 (ArithTerm.succ
(natToTerm (s := s) k'))

       (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A)
BetaForm_sh))))

      (substForm 5 V.val IsSuccCode))) =

    and_form

     (substForm 3 V.val (substForm 2 (natToTerm (s := s) k')

      (substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A)
BetaForm_sh))))

     (and_form

      (substForm 3 V'.val (substForm 2 (ArithTerm.succ (natToTerm (s := s) k'))
```

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))))

(substForm 6 V'.val (substForm 5 V.val IsSuccCode))) := by

dsimp [and_form, substForm, substTerm]

rw [substForm_fresh_of_NotFreeForm _ 6 V'.val (NotFreeForm_C4_V A B k' V 6 (by decide) (by decide) (by decide) (by decide) (by decide) (by decide))]

rw [substForm_substForm_term_comm _ 3 6 V'.val h_occ6_C5]

rw [hV'_simp] at hV'

have h_beta_V := and_elim_left_Δ h_max h_con _ _ hV'

have h_rest := and_elim_right_Δ h_max h_con _ _ hV'

have h_beta_V' := and_elim_left_Δ h_max h_con _ _ h_rest

have h_is_succ := and_elim_right_Δ h_max h_con _ _ h_rest

-- 5. Aplicar la funcionalidad de Beta (Paso 1.3)

let k'_closed : TermClosed s := ⟨natToTerm (seq_code k'), isClosed_natToTerm (seq_code k')⟩

have h_V_eq_k' := beta_sh_functional_closed Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ A B k' k'_closed V ih_k' h_beta_V

have h_V_eq_symm := eq_symm_Δ h_max h_con h_V_eq_k'

-- 6. Sustituir V.val por natToTerm (seq_code k') en h_is_succ con conmutación

have hfree_succ1 : IsFreeFor 5 V.val (substForm 6 V'.val IsSuccCode) := IsFreeFor_closed _ 5 V

have hfree_succ2 : IsFreeFor 5 (natToTerm (seq_code k')) (substForm 6 V'.val IsSuccCode) := IsFreeFor_closed _ 5 k'_closed

have h_comm_V : substForm 5 V.val (substForm 6 V'.val IsSuccCode) =

substForm 6 V'.val (substForm 5 V.val IsSuccCode) :=

subst_comm_closed IsSuccCode 5 6 V V' (by decide)

have h_comm_num : substForm 5 (natToTerm (seq_code k')) (substForm 6 V'.val IsSuccCode) =

substForm 6 V'.val (substForm 5 (natToTerm (seq_code k')) IsSuccCode) :=

subst_comm_closed IsSuccCode 5 6 k'_closed V' (by decide)

have h_is_succ_in : substForm 5 V.val (substForm 6 V'.val IsSuccCode) ∈ Δ := by

rwa [h_comm_V]

have h_is_succ_ded := SPA_Deduction_Internal.eq_subst Δ 5 (substForm 6 V'.val IsSuccCode) V.val (natToTerm (seq_code k'))

(SPA_Deduction_Internal.hyp Δ _ h_V_eq_symm) (SPA_Deduction_Internal.hyp Δ _ h_is_succ_in) hfree_succ1 hfree_succ2

have h_is_succ_num_comm := maximal_consistent_closed_under_deduction Δ h_max h_con _ h_is_succ_ded

have h_is_succ_num : substForm 6 V'.val (substForm 5 (natToTerm (seq_code k')) IsSuccCode) ∈ Δ := by

rwa [h_comm_num] at h_is_succ_num_comm

-- 7. Aplicar la representación de IsSuccCode

have h_vk1_def : seq_code (k' + 1) = pair 4 (seq_code k') + 1 := rfl

have h_V'_eq := (IsSuccCode_representable Δ h_max h_con h_henkin h_add_zero h_add_succ h_mul_zero h_mul_succ (seq_code k') (seq_code (k' + 1)) h_vk1_def).2 V' h_is_succ_num

-- 8. Sustituir V'.val por natToTerm (seq_code (k' + 1)) en h_beta_V'

let Beta_k1 : ArithForm s := substForm 2 (ArithTerm.succ (natToTerm (s := s) k'))

(substForm 1 (natToTerm (s := s) B) (substForm 0 (natToTerm (s := s) A) BetaForm_sh))

```
  let k'1_closed : TermClosed s := ⟨natToTerm (seq_code (k' + 1)), isClosed_natToTerm (seq_code (k' + 1))⟩

  have hfree_b1 : IsFreeFor 3 V'.val Beta_k1 := IsFreeFor_closed _ 3 V'

  have hfree_b2 : IsFreeFor 3 (natToTerm (seq_code (k' + 1))) Beta_k1 := IsFreeFor_closed _ 3 k'1_closed


  have h_final_ded := SPA_Deduction_Internal.eq_subst Δ 3 Beta_k1 V'.val (natToTerm (seq_code (k' + 1)))

   (SPA_Deduction_Internal.hyp Δ _ h_V'_eq) (SPA_Deduction_Internal.hyp Δ _ h_beta_V') hfree_b1 hfree_b2

  exact maximal_consistent_closed_under_deduction Δ h_max h_con _ h_final_ded



end SPA
```